The CB Programming Language

Version: July 5, 2020

Table of Contents

Copyright

Author

This programming language has been designed by John T. Bagwell Jr. of Sandpoint, Idaho. Bagwell had experience as a developer of compilers and supervisor of a team of compiler developers before retirement. He also published and presented a paper on code generation with local optimization in compilers in 1972.

The CB Programming Language

CB is a language designed for general application programming or server scripting. It is a complete programming language, not just for scripting, designed to be compiled. It is a pure object-oriented language, because all values are objects, and is designed for simplicity of use. Some typical language features in other programming languages are changed in order to achieve lower error rates in writing programs.

If used as a scripting language, CB can be used in creating web sites and in creating tools to run under a server such as Apache. It can also be used for creation of stand-alone programs. A server environment simplifies input and output and simplifies user interaction. There is no standard graphical library.

CB is intended to be compiled. The source files as a set create a library system which is managed by the compiler and development tools.

After compiling, the result is stored in a file which produces the final program for execution. A comparison is made for source file date being newer than file date to invoke the compiler as needed.

Some of the Features of CB

- Every value is an object. Constants, arrays and expressions are objects. Procedures and names of types are not objects.
- Procedures (methods and functions, properties, type casts, ...) are applied to a base object. For example, the square root of **X** is **X.sqrt** rather than **sqrt(X)**. Methods with no base type specified have no base.
- All variables and arrays are declared and typed. It is a statically typed language, meaning variables and functions (etc.) are explicitly typed. It has implicit type conversion rules, not requiring type casting for built-in types.
- Limited inferred typing is used, for example in the **for** construct and initial values for arrays.
- Default access to object members is public.
- Procedure overloading and overriding are supported. Operator definition is included. It supports subtyping polymorphism. A procedure can have different implementations for different parameters, using parametric polymorphism.
- The case of all reserved words is lower case.
- A concern in the design is type safety. There should be fewer opportunities to confuse values, like types being misused.
- Another concern is maintainability. The language is concise. Making source edits easier is one goal in the language design.
- Arrays and strings are designed to avoid overrun and bounds violations for safety and a degree of protection against hacking.
- Arrays are mapped, a matrix is fixed-dimensioned.
- Arrays, by default, use a balanced tree implementation to improve search and insert performance. Alternate storage methods can be specified.
- An array can have string-valued indexes rather than being restricted to integer indexes.
- Arrays of arrays are possible. This is different from a two-dimensional array, also available as a matrix.
- A number of array procedures are provided, based on an array and possibly returning an array. Simple array expressions are permitted, allowing optimized inline compiled code.
- The language is expandable. For example, types complex and string are defined object types; an existing definition is provided.
- Pointers are not used. In some cases a choice between value and reference is provided.
- CB is a late-binding language.

- Less punctuation is used than in the C family (C, C++, C#, Java, JavaScript, …, PHP). There is less use of parentheses. Some operators are different; equality is "**=?**" rather than "**==**" and inequality is "**\=**" rather than "!=". Semicolon is not used to end or separate statements.
- A generator function enables obtaining a sequence of values as if they come from an array.
- Data structures with repeating parts and a single descriptor part can be defined as structures. Lists, stacks, trees, queues, chains, rings, hash tables, etc.
- A user-defined list or tree can also be made to operate like an array using a structure.
- Functions and objects can support multiple types with a single source definition.
- A function or method invocation with no parameters does not use a pair of parentheses.
- An enhanced object type can keep units apart, like Fahrenheit versus Celsius, or Meters versus Inches.
- The model for file and directory manipulation is different, more object-oriented. It is defined in a set of objects.
- Formatting for output is designed to reduce errors. This is not part of the language, but is a set of objects and procedures in a library.
- Operators, complex type, array storage, string storage and type casting can be defined or modified.
- Some reflection features are included.

Omitted Language Features

These are some of the omitted or simplified features:

- Different sizes (or precision) of floating point. One precision (double) is provided.
- Pointers and pointer arithmetic are omitted because of danger and misuse.
- Garbage collection is used automatically.
- Type punning, or union, is omitted because of danger and misuse.
- Namespaces, modules, packages, etc. are not defined. Simple scope rules are used.
- Multiple statements per line are disallowed. This simplifies editing and maintenance.
- Events and event handling are left to library definitions.
- Threading and parallel computation are not implemented.
- Prefix ++ and -- operators are omitted; postfix ++ and -- are included, except for complex type.
- The C language family ternary operators '?' and '**:**' which allow a choice in the middle of an expression are omitted. An alternative is provided using an inline **if** in the expression.
- Direct input from a user is omitted; there is no graphic support. Programs run on a server or as background tasks or use an imported library or API.
- Closures, functions as parameters, callbacks, delegates and lambda expressions (these are all related features) are not defined.
- Error handling is not part of the language syntax.

Source File Format

Source files are text files in the UTF-8 code. Output strings to text files and database values and printed strings are by default in UTF-8 encoding. A UTF-8-encoded byte order mark (hex EF BB FF) is accepted and ignored if it is the first Unicode code point in the source text.

The end-of-line in the source is the Unix form using LF (linefeed) or the Windows form using the sequence CR LF (return followed by linefeed, 0x0C 0x0A). Either of these forms can be used equally, regardless of platform. In other words, CR is ignored before LF.

UTF-8 characters other than those identified in this document are valid only in character string values.

Source file lines are limited in length to 5,000 bytes. Source lines cannot be continued.

A source file consists of object and structure definitions, typeset definitions, valset definitions, layout definitions and pragma statements.

There are no executable statements, control constructs, or assignments outside of a procedure.

The main program is defined in the object named @Main, the constructor is the program.

There are no multiple statements on a line except for simple if statements.

Comments follow a # sign, to the end of the line. The comment mark acts as an end of line. There are no multi-line or embedded (block or inline) comments.

A space must be placed between source tokens if the meaning requires a space.

Programs are expected to use indentation for readability. Readability suggestions -

```
Surround assignment operators with blanks.
Place a blank after a comma or a left curly brace.
Indent construct inner lines by at least 2 or 3 positions.
Use lots of comments.
```

The language does not use semicolons, except in string values.

Libraries

The library inclusion statement format is:

pragma use *library_spec* [,...]

where *library_spec* is a specifier of a library and a file, as a name list of files. Keyword **pragma** is reserved, words after **pragma** are not reserved.

There are some standard library files which are automatically included, in directory **std/**. These define standard objects or procedures and constants. The directory names are not reserved.

The *library_spec* has the form: **directory/member** where **directory/** can be omitted; a search is then done for the member in the directory **user/** then the directory **std/**. Subdirectories are permitted and searched.

The *library_spec* cannot be a string expression. Quote marks are not used.

Standard libraries are implicitly included, as needed. They are:

Table: Standard Libraries

Name[.cb]:	Defining:
std/array	Array and matrix definitions
std/complex	Complex data type definitions
std/structure	Structure and array definitions
std/directory	Directory procedures
std/files	File procedures
std/fmt	Formatting output
std/io	File operations
std/math	Mathematical constants and functions
std/settings	Configuration file
std/std	Miscellaneous procedures etc.
std/string	String procedures and structure definition
std/typesets	Typesets

Terminology

The word "object" in this language and this language description is used in an atypical way. Most object-oriented programming languages (OOPLs) use the word "class" instead as the keyword for the definition of a new object type.

These terms are clarified:

- ***object*** - the traditional object, an encapsulated entity containing data and associated functions and methods. In CB, every data item or value is an object, even the basic things like constants and variables of basic types. These objects have predefined functions.
- ***object type*** - the new type which is defined by the construct with the word **object**. This is actually a "class" in common programming language terminology.
- ***enhanced object type*** - an object definition can inherit from a basic type, such as float, and enhance the meaning of that type. This allows a way to add attributes or restrict usage.
- ***basic type*** - the predefined types **int**, **uint**, **float** and **bool**.
- ***selector*** - the period character ("**.**"). The left side is an object or an object type name, the right side is a member, type name, built-in procedure or many other uses.
- ***member*** - every item contained in an object is a *member* of that object. Also a single value in an array is an array member, or array *element*. Lists also have members.
- ***index*** - the subscript of an array. It can be an int, uint or string. The position identification for an array element.
- ***statement*** - a line or construct which defines an action.
- ***construct*** - a multi-line statement.
- ***declaration*** - a line or construct which defines a type or defines a procedure. This is not an executable statement.
- ***block*** - a sequence of statements in a construct. A block acts as a statement.
- ***Lvalue*** - an item which can be assigned a value in an assignment. A 'left-hand' value, including a put property.
- ***reference*** - a parameter or base object passed without copying the value, using a created pointer.
- ***value type*** - a type which is passed by copy. These types are uint, int, bool, float and valset.
- ***array -*** a structure with indexed members. Two forms are supported: associative (mapped) and ***matrix***.
- ***matrix -*** a one- or two-dimensioned array with fixed allocation of space in contiguous memory. A matrix can be dynamically allocated or with constant dimensions.

- ***curly brace*** - "**{**" or "**}**". Used for defining constructs and blocks.
- ***square bracket*** - "**[**" or "**]**". Used for array indexing and substrings.
- ***structure*** - a set of values, possibly indexed, ordered or unordered. An array and a string are a kind of structure. A structure contains a **anchor** part, which may represent a root, and multiple **leaf** parts. New structure types can be added by defining them in an object type definition with the option **struct**.
- ***layout -*** a type which describes the parts of an integer. Useful also for extracting bit fields without need for a shift operator.
- ***procedure*** - a method (a "subroutine") or function, defined inside an object or with no base type or on a basic type. In some languages these concepts may be called methods or messages. Procedure definitions include:
  - ***method*** - a procedure which does not return a value, thus it cannot be used as a primary element of an expression. It is used as a statement.
  - ***function*** - a procedure which returns a value, usable in an expression.
  - ***property*** - a pseudo-variable, which may be read-only or write-only or both read and write. Properties are defined as get properties and put properties. A get and put can share the same name.
    - ***get property*** - defines getting private data as a public value. Also a read-only variable.
    - ***put property*** - defines storing into an Lvalue, with appearance of a write-only variable.
  - ***generator function*** - a function which behaves like an array in a **for** construct.
  - ***defined operator -*** operators +, -, *, /, etc., can be defined on objects.
  - ***type casting -*** after a selector "**.**" a type name "casts" an object to a different type.
  - ***constructor -*** a procedure named the same as the object type, used to create new items of that type.
- ***access*** or ***accessibility*** - the level of access to a member of an object. One of these levels applies:
  - ***private*** - visible and accessible only in the object and its procedures and functions, not in inheriting objects. Defined with the option **private**. In a block, this access prevents visibility in a nested level.
  - ***shared*** - visible and accessible in the object and its procedures and functions, also in inheriting objects. Defined with the option **shared**.
  - ***public*** - default if access not specified with a keyword. Accessible anywhere.
- ***named constant*** - a name and type can be assigned a constant value.
- ***parameter*** - a value passed to a procedure. The base is also implicitly a parameter.

- **shape** - a structure which has a name and which defines storage access.

Reserved Words

Reserved words cannot be used to define names for a program; they have predefined uses in the CB language. Reserved words are also called keywords. Reserved words are in lower case. Other case forms of these words are not reserved words but they should be avoided.

There are 47 reserved words.

Table: Reserved Words

`$`	`complex`	`for`	`leaf`	`pragma`	`static`	`uint`
`abstract`	`const`	`from`	`method`	`private`	`string`	`until`
`anchor`	`do`	`func`	`new`	`property`	`struct`	`uses`
`and`	`else`	`gives`	`nil`	`put`	`switch`	`valset`
`bool`	`false`	`if`	`not`	`repeat`	`then`	`while`
`break`	`final`	`int`	`object`	`return`	`true`	
`case`	`float`	`layout`	`or`	`shared`	`typeset`	

The single dollar sign (**$**) is a reserved word, used in procedure definitions, meaning the current object's content. It is similar to the word *this* or *self* in other programming languages.

Some reserved words (**final**, **return**, etc.) have multiple uses.

Names

A *name*, or *identifier*, must begin with a letter. Digits or letters (in either case) are allowed after the initial letter. The case of letters is significant. Spaces shall not be used in a name.

Also considered as a "letter" for use in a name are the characters dollar sign (**$**), underscore (**_**), at-sign (**@**) and exclamation mark (**!**). An initial underscore or an initial at-sign are used for language-defined names; user-defined names should avoid these as initial characters.

Legal names: inTheMiddle, a1, time_worked, days@work, $cost.
Invalid names: 1A, two words, A&B, x-hyphen, ¥999, 2×4.

Special Names ! and @

The single exclamation mark (**!**) is a supplied mathematical function, computing a factorial. For example, the expression **4.!** has value **24**. It is not a reserved name.

The single at-sign (**@**) is a method name, with no base type specified, with one string parameter. It outputs the string with end-of-line characters. It is an "echo" or "print" method, customarily pronounced "out". It is not a reserved name.

Main Program

A "main program" begins by implicitly constructing and instantiating an object with the *type_name* @Main, in effect, executing a line:

```
@Main
```

This invokes the constructor of the object type which is named @Main. This name is defined in a setting.

Settings

The settings (configuration) data is found in library **std/settings.cb**. It consists of a pragma line:

```
pragma settings
```

followed by *command* lines of this form:

```
command value1 value2 …
```

For example, it contains these lines, defining the format of displayed numbers:

```
locale thousands ','
locale decimal_point '.'
```

The following lines are among the defined types and array indexing defaults:

```
type complex @Complex
type string <@String>
array <@Array>
matrix <@Matrix>
start @Main
```

A modified settings file is permitted. It will contain overrides to the standard settings.

Blocks and the Scope of Names

A *block* is a sequence of statements and declarations which delimit the scope of meaning for names. A block is also called a *construct*, since it usually consists of multiple lines.

A name has a scope limited to the object it is in when **private** is specified, or to the *block* in which it is defined. The scope begins where the name is defined. At the end of that block, the scope is terminated and the variable, array, string, or object is erased and not visible.

Within a block, a name cannot be redefined except in a new scope level, a nested block.

All names defined outside an object have global scope, available anywhere. Data (variables and arrays) are not allowed outside of an object.

Names of object types, structures, valset types, layout names, typeset names are in the same name category and a name must be unique for those usages.

Data Types Supported

The standard data types in CB are integer, unsigned integer, float and bool. They are also called basic types. Type **complex** is defined by a line in the settings as an object. Alternate definitions are possible.

New types can be added as objects.

Integer

Integers have no fractional part, and are (by default) signed 4-byte integers, with type keyword **int**. Integers of other sizes can be declared. They are declared as **int.16**, **int.32** (also as **int**) and **int.64**. There are no **int.1** or **int.8** types. All of these are signed. The appended number is the number of bits used.

Unsigned integers are also available, as **uint.64**, **uint.32** (also as **uint**), **uint.16**, **uint.8** and **uint.1** types. The size numbers after the period are bit lengths.

Alternate names can be used as shown. Both **uint** and **int** types are integers.

Table: Integer Value Ranges

Type	Alternate Name	Integer value ranges, shown with commas
uint.1		0 or 1
uint.8		0 to 255
uint.16		0 to 65,535
uint.32	uint	0 to 4,294,967,295
uint.64		0 to 18,446,744,073,709,551,615
int.16		-32,768 to +32,767
int.32	int	-2,147,483,648 to +2,147,483,647
int.64		-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Float

Floating point values are based on the IEEE 754-2019 standard, implementing double precision binary floating point arithmetic, using 64 bits (8 bytes.) There is no shorter (standard or half float) or longer form. The decimal floating point in that standard is not implemented.

There are no constants denoting the IEEE-754 infinity and not-a-number (NaN) or subnormal values for the float type. These may have multiple internal values.

The type supported is **float**. The size is 64 bits. A **float** constant is a floating point value that consists of a numeric value where each digit in the value is from 0 to 9, a required decimal point, a required digit on each side of the decimal point, and an optional exponent after a letter **E** or **e** that indicates a multiplier by a power of 10. A float has about 18 digits, with an exponent maximum of about 308.

In addition, the identifier **_PI** is defined as a float named constant with the value pi (π) to full float value in the automatically included math library. This identifier can be redefined; it is not reserved. Also defined are **_E**, as constant ***e*** and other values.

Bool

The type **bool** is a logical value, with true or false values. It is named for George Boole, who defined logic manipulation rules. It is also called Boolean or Logical in some programming languages.

The result of a comparison or bool expression is either **true** or **false**. A bool value occupies one byte in storage. The constant true has value 1 and false has value 0 internally.

A bool value in a layout can occupy a single bit.

The **if** statement and other places require a bool expression.

The basic arithmetic types may be type cast to **bool**. A nonzero value becomes **true**, a zero value becomes **false**.

A bool value may be type cast to an integer or unsigned integer, with **true** becoming 1 and **false** becoming 0. It may be type cast to a string; **true** becomes 'T' and **false** casts to 'F'.

String

The string type definition is defined as an object type in the library **std/string**. The library is implicitly included.

Strings vary in length. The length is from 0 to a value that fits in **int.32**, approximately 2 billion bytes. The string type in CB is <u>not</u> terminated by a "null character" (\00) as in C/C++ and other languages. Characters in a string are UTF-8 code, using 8-bit bytes. Strings are immutable; internal character values cannot be modified.

There is no separate type for single characters. There are no fixed-length strings.

Valset

Valset items are named constant unsigned integer values, grouped into a new type name. This is similar to the C language 'enum', but more restrictive because the names are typed. A valset constant name can be used only as a value in a variable or array using the correct valset type name as its type.

Complex

The complex type definition is found as an object type in the library **std/complex**. The library is implicitly included. The type name **complex** is a reserved word mapped in the settings file to the name **@Complex**.

Complex type is a pair of float numbers, one real and one imaginary. Complex variables are immutable; a part of the value cannot be altered.

Alternative implementations can be defined. Arithmetic operators supported are the same as type float, except for remainder and postfix ++ or --. Comparisons supported are equal and not equal.

Table: Mathematical Functions

Function(y):	Base(x) Type:	Result Type:	Returns:
abs	float int	(base)	Absolute value
arccos	float	float	Arc Cosine
arccosh	float	float	Arc Hyperbolic Cosine, **x**: [1,+infinity)
arcsin	float	float	Arc Sine
arcsinh	float	float	Arc Hyperbolic Sine, **x**: (-infinity,+infinity)
arctan	float	float	Arc Tangent
arctan(float **y**)	float	float	Arc Tangent of **y**/**x, y** is float
arctanh	float	float	Arc Hyperbolic Tangent, x: (-1,+1)
Binary	(any uint or int)	string	Bits in the integer as a string
bitSize	(any uint or int)	uint	Returns 1, 8, 16, 32 or 64
ceil	float	float	Whole number > or =, away from 0
cos	float	float	Cosine
cosh	float	float	Hyperbolic Cosine
exp	float	float	Exponential, **e** to the power
floor	float	float	Whole number, truncated toward 0
Hex	(any uint or int)	string	To a hex string, no leading 0x
isEven	int uint	bool	Returns **true** if even else **false**
isInfinity	float	bool	Test for "Infinity" + or -
isNaN	float	bool	Test for "Not a Number"
isNeg	int float	bool	Returns **true** if < 0 else **false**
isNonZero	int uint float	bool	Returns **true** if \= 0 else **false**
isOdd	int uint	bool	Returns **true** if odd else **false**
isPos	int float	bool	Returns **true** if > 0 else **false**
isZero	int uint float	bool	Returns **true** if =? 0 else **false**
log	float	float	Logarithm, base ***e***
log10	float	float	Logarithm, base **10**
maxValue(**y**)	float int	(base)	Maximum of base(**x**) and **y**
minValue(**y**)	float int	(base)	Minimum of base(**x**) and **y**
round(uint **y**)	float	float	Round **y** digits away from zero
roundEven(uint **y**)	float	float	Banker's rounding, even lowest digit
sin	float	float	Sine
sinh	float	float	Hyperbolic Sine
sqrt	float	float	Square root
tan	float	float	Tangent
tanh	float	float	Hyperbolic Tangent
!	uint.8	uint.64	Factorial

The standard math functions are defined in standard library **std/math**.

Functions **maxValue** and **minValue** require the base and **y** to be the same type.

The following functions (or get properties) are defined for complex type:

Table: Complex Mathematical Functions

Function:	Result Type:	Returns:
abs	complex	Absolute value
arccos	complex	Arc Cosine
arg	complex	Argument, same as theta, the angle for polar form
arcsin	complex	Arc Sine
arctan	complex	Arc Tangent
Conjugate	complex	Complex conjugate
cos	complex	Cosine
cosh	complex	Hyperbolic Cosine
exp	complex	Exponential, **e** to the power
Imaginary	float	Imaginary part
isNonZero	bool	Returns true if \= 0 else false
isZero	bool	Returns true if =? 0 else false
log	complex	Logarithm, base ***e***
log10	complex	Logarithm, base **10**
Real	float	Real part
Reciprocal	complex	Complex 1.0/x
sin	complex	Sine
sinh	complex	Hyperbolic Sine
sqrt	complex	Square root
tan	complex	Tangent
tanh	complex	Hyperbolic Tangent
xI	complex	Multiply by the imaginary ***i***

Function **isZero** is **true** if both real and imaginary parts are zero.

Function **isNonZero** is **true** if either real or imaginary is nonzero.

Operators for arithmetic **+**, **-**, *****, **/** and unary **+**, **-** are defined. Postfix operators **++** and **--** are not.

A complex constant or value is written as *complex(real part, imaginary part)*, using the constructor. The parameters are named Real and Imaginary and have default values of 0.0, so the constructor can be invoked using the optional parameter names Real and Imaginary, as in this example:

```
complex Ix3 = complex(Imaginary=3.0)
complex Z = Ix3.sqrt # example function usage
```

The following public constant names are defined:

```
_COMPLEX_I         the imaginary value i
_COMPLEX_ONE       real part 1.0, imaginary part 0.0
```

Value Types and Reference Types

Value types are simpler data types which pass a copy of a value to a procedure, and which copy a value when they are assigned. These are the builtin or basic types except string and valset typed values.

Reference types are more complicated. When reference types are passed to a procedure, a copy of a reference (a hidden pointer) is passed. When an assignment is done, the reference is used, rather than the value. These are arrays, strings and typed objects.

Two references can refer to the same copy of data. If one item is altered, both references see the same altered data.

Typed objects provide some control over this behavior; also procedures may force a specific copy or reference behavior on their parameters.

In an object definition, a name of a reference type item is always treated as a reference.

The Value Nil

The reserved word **nil** identifies a reference to nothing, a reference value for any object type. It does not mean an uninitialized object; it is a "missing" object.

A function can return **nil** to indicate "no object". The default value of a defined type object reference is **nil**.

The value **nil** can be assigned to a reference.

Objects and Inheritance

Every value is an object and has an object type.

Objects support single (multi-level) inheritance.

Arrays and matrices and strings implicitly inherit a defined structure type.

Selection

The selector operator, a period, is used to refer to a member (variable, procedure, etc.) of an object, or to specify a type cast.

For example:

> **obj.fun** invokes the function f**un** on object **obj**.
> **var.string** uses a type cast to convert variable **var** to a string.
> **type.member** refers to a static entity **member** in the static area of object **type**.

Expression Operators

Table: Operators in Expressions

Operator	Description	Priority	Associativity	Operand Types
.	Member/procedure / type cast / name selector and more	11	n/a	n/a
++	Postfix, incrementation	10	n/a	int
--	Postfix, decrementation	10	n/a	int
+	Unary prefix +	9	n/a	int float
-	Unary prefix -	9	n/a	int float
\	Unary prefix bitwise complement	9	n/a	uint
^	Exponentiation	8	right	int float
*****	Multiplication	7	left	int float
[*]	Matrix Multiplication	7	left	int float (2D matrices)
/	Division	7	left	int float
/	Split string on a separator	7	left	string
%	Remainder / Modulus	7	left	int float
&	Bitwise and	7	left	uint
|	Concatenation of strings	7	left	string
+	Addition	6	left	int float
-	Subtraction	6	left	int float
|	Bitwise inclusive or	6	left	uint
~	Bitwise exclusive or	6	left	uint
<:	After	6	left	string
:>	Before	6	left	string
=?	Equality comparison	5	left	(any)
\\=	Not equal	5	left	(any)
<	Less than	5	left	int float string
<=	Less or equal, not greater than	5	left	int float string
>	Greater than	5	left	int float string
>=	Greater than or equal, not less than	5	left	int float string
not	Logical (bool) NOT (unary prefix)	4	n/a	bool
and	Logical (bool) AND	3	left	bool
or	Logical (bool) OR	2	left	bool
=	Assignment	1	right	[special rules]
<->	Swap operator	n/a	n/a	[special rules]

Type **int** above includes **uint** and any valid size. Type **uint** in the table includes any size.

Highest priority numbers are evaluated first.

Assignments do not produce a value. An assignment is a statement, not an expression.

The exponentiation operator **^** is right associative; the expression **a^2^3** is the same as **a^(2^3)**. The exponent must be a small int value, -63 to +63.

The comparison (relational) operators (**=?**, **\=**, **<**, **<=**, **>**, **>=**) return a **bool** value **true** or **false**. The values **true**, **false** and **nil** cannot be compared with anything. Thus, the comparison expression **a < b < c** is invalid.

Division of an integer by an integer produces a truncated (toward zero) integer result. Division by zero causes an error, **DivideByZero**.

The remainder (modulus) operator **%** returns the remainder from integer division. For float operands, the division is truncated to an integer. For **z = x % y**, the sign of the remainder is the same sign as **x**. The right operand y shall not be 0. The result **z** is **z = x - (x / y) * y**, using integer division.

The swap operator **<->** requires both sides to be an Lvalue and the same type.

The string *after* and *before* and *split* operators are described in the String Operaators discussion, later in this document.

Conditional Values (Inline If)

Within an expression, a simplified inline **if** can be used to choose between values based on a comparison or **bool** value. This is similar to the question mark ternary operator in C-like languages. A required pair of parentheses enclose an **if** and **else**, in this form:

(if *bool_expression* **then** *true_value* **else** *false_value* **)**

The true and false values must be expressions of the same type. Example:

```
xyz = (if abc.isPos then abc else -abc) # equivalent to xyz = abc.abs
```

Postfix Operators ++ and --

Postfix operators ++ and -- require the left operand to be an int or uint Lvalue. An object reference to an element completes the reference, returns that value, then increments it.

The postfix operators ++ and -- increment or decrement the left operand by 1 and return the original value. Thus, if mm has the value 4, mm++ changes mm to 5 and returns 4. This operator as a statement also acts as an assignment statement, ignoring the original returned value. These can be statements because they alter a value.

These operators cannot be used with float or complex type.

There are no prefix versions of these operators.

Unary Operators

The unary operators **+**, **-** and **~** are restricted in placement. They shall not follow a binary or unary arithmetic operator or a relational (comparison) without a space. They may follow a left parenthesis, a left square bracket, an assignment operator, a comma, or the logical operators **and**, **or**, **not**.

Shortcut Operators

The operators **and** and **or** evaluate the right side only as necessary. The left side determines the value alone for some cases:

Table: Shortcut Operators

Left Side	Operator	Right Side	Result
true	**or**	skipped, not evaluated	left side alone determines value = **true**
false	**or**	evaluated	the value is the right side value
true	**and**	evaluated	the value is the right side value
false	**and**	skipped, not evaluated	left side alone determines value = **false**

This skipping is called "short circuiting" or "shortcut." Any functions or postfix ++ or -- in the right side which were skipped may have had side effects which will not be done.

Short circuiting is not applied in a bool assignment.

The bit-wise operators **&** and **|** do not short circuit.

Variables and Array Declarations

A variable is a single value of any type, declared:

type name_list

Example:

```
int abc
string name, last_name, nick_name
```

It has a *type*, which is **bool**, **int**, **uint**, **string**, **complex**, **float**, or a **valset** type name or an object type name.

An initial value can be assigned in the declaration:

int maxSize = 1000

Structure declarations are also allowed as a *shape*:

string<List> titles = ['duke','baron','viscount','earl','prince']

Within a declaration, only one type can be named. This is invalid usage:

int ijk, string str

Implicit and Explicit Type Conversions

An assignment will convert the type of the right side to the type of the left side if it can do so.

Unsigned **uint** widens as it converts to **int**. In other words, **uint.16** widens to **int.32**.

Conversions to and from the numeric types int and float are obvious. Overflow or loss of significant digits in converting from float to int is ignored.

Conversion from a numeric value to string is allowed by type casting or implicit conversion, and is the same result used in the standard method **@**, or the same as a value insertion in a string. Integers are exact, float numbers are approximate.

Conversion from a string to a numeric value will cause an error if there is no valid conversion after trimming trailing blanks. An empty string and a value too large to be an **int** both cause a conversion error.

A string value like "123GO" does not convert to the integer 123. It causes a conversion error.

Integer Arithmetic

Integers with widths 32, 16, or 8 bits or 1 bit (which is unsigned only) "widen" to the next size in an expression, when combined with +, -, or * operators. There is no widening for division or for the bit-wise operators.

When a longer integer is assigned to a shorter integer, the upper part is discarded without error; integer overflow or truncation is not diagnosed.

Widening or shortening (truncation) can be specified by type cast; abc**.**int**.**16 extracts the least significant 16 bits of the integer abc.

An integer divided by an integer truncates toward zero. It does not yield a float result.

Bool Expressions

A **bool** expression has a value of **true** or **false**. These are also reserved words for the values. Arithmetic on these values is disallowed; the value of **true** is nonzero but not specified. A non-bool value cannot be compared with **true** or **false**. Bool values cannot be compared with **=?** or **\=**. A similar result is obtained by use of **and**.

A bool expression cannot be cast into a float value.

The bool operators **and** and **or** in a bool expression evaluate using shortcut evaluation except in an assignment. If a value can be determined while skipping a subsequent portion of the expression, the skip is permitted. This shortcut evaluation does not apply to the bitwise and (**&**) and bitwise or (**|**) operators.

One example:

```
if abc <= 9 AND def > 0 break # "def > 0" is skipped if abc > 9
```

Type Casting

A value can be converted to a compatible type by casting the type:

value**.***type_name*

converts the type or changes to a compatible type. If the value can be assigned to an item of the *type_name* specified, it is a legal type cast. This is not "type punning" which retypes without conversion. There is no type punning in CB.

An object variable can be cast as a type it inherits. The result loses any additional features.

A value that has a type cast is considered a new value, so a reference to the old value is not used.

Type casting applied to an array applies to each element, creating a new array.

Any basic type casts to string. A string casts to a type if it can be legally converted.

A bool can be type cast to string; the result is **'T'** or **'F'**. Casting to integer from **bool** is allowed; **true** casts to 1, **false** to 0.

Casting to **bool** is not allowed. Comparison operators and the inline **if** can be used.

Casting between **int** and **uint** - the sign may change to a data value, and vice versa, without an error signalled. Data loss can occur if the size of an integer value is shortened, also with no error. Integer sizes remain part of the type; **abc.uint.8** extracts the low-order byte. Casting from **int** to **uint** may convert a sign to a data value, and vice versa.

Casting from **float** to **int** or **uint** is allowed. There is no test for loss of significance.

Data Values

Named data can be any one of these things, all of which are considered objects:

- A simple item (a "scalar") with a basic type **bool**, **float**, **int**, **uint**, **string**, **complex**, or a **valset** type.
- An array has integer values or string values as indexes. The first index is 0 when unsigned integer indexes are used. An array element can be missing (**nil**), or undefined. A missing numeric value has the value 0, and a missing string value is the zero-length string value '' (two apostrophes.)
- A structure.
- An object, which may have procedures and functions and other definitions in it. The definitions may be marked **private** or **shared** to control access. The *object_type* is defined with an **object** declaration and variable or array names or functions can be assigned to that type. An object can inherit.
- A named constant, which has a type.

Constants

Integers (**int** or **int.32**) are signed integer values between -2147483648 and 2147483647. The sign is not part of a constant, it is a unary operator **+** or **-** in an expression. A 16 bit integer constant has a suffix of **S16** or **s16** and an 8 bit integer has a suffix of **S8** or **s8**. An unsigned integer constant has a suffix of **U** or **u** and a bit length.

An unsigned or signed integer constant can also be written as 0X*dddddddd* where each *d* is a hexadecimal digit. The hexadecimal digits A through F and the X can be either case. There are up to 8 hexadecimal digits in a **uint.32**; there are 2 hex digits per byte. Length and sign suffixes with **U** or **S** are optional. Otherwise it is assumed unsigned. 0X80000000 is the largest integer negative value. Examples: 0xFFs8, 0XabcdU16, 0xCD.

There is no support for octal values. A leading zero is ignored in a numeric constant.

A binary constant can be used for signed or unsigned integers. It has the form 0**B**ddd...d where each bit **d** is a bit of value 0 or 1. The **B** can be **b**. A sign or unsigned suffix (**S** or **U**) with optional bit size can be on the end. By default, a binary constant is unsigned. The value has extra leading 0 bits added at the left (high end) as needed. At most 64 bits are allowed.

Float constants have a decimal point and an optional exponent after an **E** or **e**. A digit must precede and follow the decimal point.

A string constant is enclosed with apostrophes or quotation marks. If it begins with apostrophe, an embedded apostrophe must be preceded by a reverse slash. Similarly a quotation mark inside a string constant started with a quotation mark must have a preceding reverse slash.

A simple string constant begins with a grave accent and ends with a blank or end of line or comma or right parenthesis.

Arrays

An array is a collection of data with a unique index associated with each array member. Arrays are indicated by specifying the form of the index in square brackets after the type.

Arrays are implemented in two forms, an associative array (a mapped structure) and a matrix, which is a fixed-dimensioned array.

An array is declared with one or more *array_spec* definitions, as follows, first the *type* of data in the array, then the index specification (shown below), then the *array_name*:

[[*index_type*] **] ...**

Associative Array

Associative array indexing (not for fixed-dimensioned arrays) is defined by a structure named **@Array**, identified by a settings file line named **array**.

The *index_type* for an associative array can be **int** (any size) or **uint** (any size except **uint.1**) or it can be **string**. The default *index_type* is **uint** (**uint.32**). An array index cannot be type **float**, **bit**, **bool** or an object type or valset type. A *typeset_name* which names only valid types can be used for the *index_type*.

When the index type is **uint** or defaulted, the minimum index is 0 for the first element. The minimum is unspecified for a (signed) **int** index; it can be negative. The maximum index or maximum number of elements of an associative array is unspecified.

Fixed-Dimensioned Array (Matrix)

A *fixed-dimensioned* array (a *matrix*) has an unsigned integer constant or an unsigned integer variable name (for a procedure parameter) as its index-type in the array_spec. These arrays have indexes which are type **uint**. They occupy contiguous space and they have default values for each element, like a variable.

A matrix specification can also be a pair of index numbers, defining a two-dimensional array or matrix. This type of array stores values by row. Thus, **X[2,3]** is next after **X[2,2]**. The minimum value of a row or column number for a matrix is 0, so the first element is **X[0,0]**.

In a procedure, if an array parameter or the base is a fixed-dimension array, it may optionally be declared with a name in each dimension rather than a constant. This name acts as a get (read-only) property returning a type **uint** value. If two array parameters in the procedure use the same name, the passed arrays must have the same dimension value for that name.

A matrix can have variable dimensions which are known values.

A row or column of a matrix can be indicated in an expression by use of functions Row or Column. For example, **X.Column(3)** is the fourth column, still a two-dimensional matrix.

Matrices with arithmetic values (types **int**, **uint**, **float** or **complex**) and having the same dimensions can be combined element-by-element using operators **+**, **-** and*****. For two dimensions, matrix multiply is supported with the operator **[*]** when the dimensions allow the multiply.

Shape

In addition, in place of an *array_spec* a *structure_id* (a structure name in angle brackets) can be specified. The combination of array_specs or structure_ids is called a *shape* specifier.

An array of arrays is permitted, by repeating another *array_spec* part, in square brackets, after the *name*. An array of associative arrays is a *jagged* array because the array lengths can vary. Two levels are allowed. This is not the same as a two-dimensional array. The outer array holds only arrays as members.

The *array_spec* or shape is optionally followed by an initial value:

= *array_value*

The *type_name* may be omitted from the array value and implied on this value to be the same as the array name's type.

Examples of array declarations:

```
float[uint] scores
string[4] suits = ['club','diamond','heart','spade']
float[][string] table # an array of string-indexed arrays of float values
```

Initially an array which is declared as shown has no elements. An empty array value is written as *type*[] or []. The type can be omitted where it is known.

An array can have as members a type of structure and vice versa. For example, an array of List structures containing strings:

```
string[]<List> name_lists # initially an empty array
```

Values can be assigned to single or multiple elements as shown:

```
int[] arr = int[100, 25, -6, 94]
arr[6] = 'XYZ'      # skipping indexes 4 and 5, remaining are unassigned
```

The index values in the initial value can be omitted or specified before the values, with a colon:

```
int[] iarr = [23,-88,10:123,345]  # defines 4 array elements, indexes 0,1,10,11
```

Values can be assigned to single or multiple elements as shown:

```
float[string] salary = ['driver':12000.00, 'sales':15000.00]
salary['CEO'] = 249995.00  # adds a new element, CEO's salary is $249,995.00
```

An array of arrays can have an initial value (all on one source line):

```
string[string][string] classroll = ['Lit':['Walt Whitfolk','Will Shakefist'],
      'Geometry':['Archie Medes','Ima Euclidian','Neva Cross'],
      'Gym':['Wilt Nicklaws']]
```

For an array of arrays, a reference with fewer indexes is a reference to an array.

A 2-dimensioned array (matrix) expects each row to have the same number of columns. An array value is extended with defaults:

```
short[3,2] Mat = [[1,2,3], [55,44], [99]] # 3 rows, 2 columns
```

An array is not instantiated unless it is assigned an initial value or an array value is assigned or a reference to another array is assigned. Declaring an array establishes a place-holder for a reference to an array.

A reference like **`Arr[]`** as a left value (an Lvalue) in an assignment, where Arr is associative and no index is given, is assumed to create an index value which is 1 higher than the maximum already used index value, or 0 if the array `Arr` is empty. This is valid only for **uint** indexed arrays.

A missing array element reference does not return a value. It must not be an operand other than an Lvalue. Existence of an array element or an object as a legal reference can be checked with the function **isNil**.

Table: Built-in Associative Array Functions

Name:	**Returns:**
Clear	method; empty the array, delete an array element
Count	the number of elements which have a value
Copy	copy of the array
Current	current element (a reference)
Each	generator function; sequences through array
First	reference to the element with the lowest index
getIndexType	string function, type name of the index
Indexes	array of index values used for elements of the array
Join(string **sep**)	string, elements as strings, concatenated with separators **sep**
Last	reference to the element with the highest index
MaxValue	maximum value in the array, must be comparable
MinValue	minimum value in the array, must be comparable
Next	moves Current, returns reference
Reset	sets Current to First
Slice(indexType **m**,**n**)	an array extracted from index **m** through **n**, **m** <= **n**

Table: Built-in Matrix Functions

Name:	Returns:
`Clear`	method; empty the matrix, set all values to default
`Column(`**n**`)`	the values of column **n**, left in place
`Copy`	copy of the matrix
`Count`	product of row and column dimensions
`Determinant`	a scalar value
`Diagonal`	diagonal values as a matrix with 1 row
`Flatten`	a single row matrix using the same values row by row
`Identity`	base is a **uint**, number of rows and cols; creates a square matrix with ones on the diagonal and zeroes elsewhere
`Inverse`	inverse of a square matrix
`numCols`	the number of columns (second dimension) as a **uint**
`numRows`	the number of rows (first dimension) as a **uint**
`Row(`**n**`)`	the values of row **n**, left in place
`Slice(`**mr**:**nr**,**mc**:**nc**`)`	the submatrix idendified
`Sum`	the sum of the values
`Transpose`	the transpose, rotated around the diagonal

Matrix values are type **int**, **float** or **complex** for functions Determinant, Diagonal, Identity, Sum and Inverse. The matrix must be square except for Sum.

Functions Column, Flatten and Row return a reference to values left in place from the base matrix.

Built-in array and matrix procedures are defined in **std/arrays**.

Array and Matrix Assignments

The only array and matrix expressions allowed are in simple assignments to arrays and method calls on them. Assignment to an array or matrix name is interpreted differently in these examples:

- abc = xyz ◀ both are arrays of the same type; copies the reference. Both arrays refer to the same data. If abc is a matrix and xyz is an array, abc is filled by rows. The counts and dimensions must match.
- abc = xyz * 2 ◀ both are arrays of the same type; the elements of xyz are doubled the assigned. Other simple expressions like this are permitted.
- abc = xyz.Copy ◀ assigns a reference to a new copy of xyz. They refer to different data sets.
- abc = xyz ◀ where the types differ, but can be converted, builds a new array and assigns a reference to it.
- abc.*method* ◀ where *method* is a method which is not specific to arrays then it applies *method* to each member of abc, or applies method to the whole array if it is an array method.
- abc = def *op* xyz ◀ where op is a binary operator and the three values are arrays of the same length and type, or matrices of the same dimensions, or one of the operands def or xyz is scalar. If they are associative arrays, the index types must match. If one element is nil, the corresponding element is ignored.
- Matrix multiply is defined only for matrix values that fit these rules for A [*] B:
  - The second dimension (number of columns) for A must match the first dimension (row count) of matrix B.
  - The result has dimensions: [row count of A, column count of B].

Examples:

```
int cnt = arr.count  # gets the number of elements in array arr
arr.sort             # sorts the values in an array with integer indexes
a_array.Clear        # removes all elements
arr[4].Clear         # removes the 5th member of arr
arr = arr + 2.0      # adds 2.0 to each element

int[2,3] X
int[2,4] A = [[2,4,6,-8],[-3,6,9,12]]
int[4,3] B = [[3,2,-1],[2,1,-4],[6,8,22],[0,7,-14]]
X = A [*] B          # matrix multiply
```

Array Values and Array Constants

An array value is a sequence of appropriately typed values enclosed in square brackets, separated by commas. A final comma may appear with no value following, before the right bracket. Index values may be supplied in the list followed by a colon and value. All the values must be the same type, and all the indexes must be the same type, with types matching the array definition.

An array of arrays will have array values in each position of the array, as required.

An empty array constant is a pair of square brackets, **[]**.

A valset array value is written like any other array value, with a list of the valset value names as the value list. All member names must be from the valset.

If all the values are constants, the bracket-enclosed array value is an array constant.

Indexes may be supplied on all values, with index, then a colon, followed by the value.

Example, initializing in a declaration:

```
float[string] prices = ['pencils':0.99, 'eraser':0.69,]
uint[] rooms = [3, 1, 4, 2]
```

The type of the array element values and indexes in an array value or constant is assumed, in most cases, from context, in a declaration. An initial value assumes the type being declared, and a value passed as a parameter or assigned to an array assumes the expected type.

If desired, the type of the array can be placed just before the left square bracket. The index types must all be the same and the index type cannot be specified in the constant.

An array value can appear in an array initialization, or an array assignment on the right. The type name must appear before the left square bracket unless it can be determined from context.

A matrix constant is like an array of arrays.

An array value cannot appear on the left of an assignment.

An array name passed as a parameter to a procedure or used as the base for a procedure is a reference to the array, not a copy.

A generator function can produce an array value. **Range** produces an array value if required.

Implementation of Array and Matrix

An array which is not fixed-dimensioned is implemented (by default) as a 'map', a balanced tree structure, using a structure name **@Array**. The indexes are maintained in sorted order. A different structure may be applied to define indexing for a specific object type.

The implementation of a matrix is defined in structure **@Matrix**.

Inline For Expression

Similar to a **for** statement, an *inline for expression* is an anonymous (unnamed) generator function which produces a sequence of values, as an array. It is a parenthesized expression of the form:

> **(for** *for_value* **from** *for_source* [*if condition*] [*return_condition*] **)**

where *for_value* is a variable of the same type as *for_source*, the type can be predefined or it is inferred, and *for_source* is an array, a string, a generator function invocation or anything allowed in a **for** statement as a source. The number of returned items from *for_source* must be limited. Only one inline for is permitted in an expression or assignment.

The **if**, when **false**, skips a value, but the *return_condition* is still tested. The *return_condition* tests which values are returned, or when to stop. It is optional, and takes one of the forms:

```
until condition
while condition
```

where *condition* is a bool expression using the *for_value* variable. The **until** and **while** determines when to stop.

Example:

```
int[] sqrs
sqrs = (for x from 26.Range(start=1))^2 # values 1, 4, 9,...,625
```

Valset Types

A new defined data type can be introduced with the **valset** declaration:

```
valset valset-type-name name-value-list
```

where *name-value-list* is a set of names with optional values, separated by commas. These values are in the form:

```
name [=unsigned_integer_value]
```

The names are names for a set of values, possibly with a constant value for the name after an equals sign. If no integer value is shown, the first name is assigned the value 0, the next is 1, etc. Once a value is assigned to a name, the next value is assumed to be 1 higher.

For each valset type, the names defined must be unique, not the same as a variable or other item, and no two names will have the same value. No overlapped values are allowed. Arithmetic on valset values is not allowed.

The value names cannot be assigned into a variable using a different valset type.

Valset constant names can be used in more than one valset type.

A valset type definition may appear in or outside an object or layout definition. It shall not be defined in a static storage area.

Valset value names are not permitted as array indexes.

Each valset declaration implicitly creates an associative array which is a static array. The array for valset VSet (for example) is named **Values** which has **string** indexes and **uint** values, addressed as **VSet.Values**.

Valset Values in a Variable or Array

One usage of valset is to declare that a variable can be assigned values from the list. The variable can then be tested for equality with a name from the valset, or the names can be used in a **switch** statement. The maximum value is 2_147_483_647, which is the maximum positive value for an **int.**

Items in a valset variable cannot be compared to integer values. An empty or uninitialized valset data value has a value of 0, which may or shall not match a named value. The function **isZero** can be used to test for zero.

For example:

```
# assigns unknown = 0, bicycle = 1, auto = 5, truck = 6 ----
valset vehicles unknown, bicycle, auto=5, truck
# declare an array:
vehicles[] rigs # takes vehicle values only, with integer indexes
rigs = vehicles[auto,truck,truck,auto] # assign 4 values from array constant
rigs[2] = bicycle # change the third value
# another example - -
valset Tstat closed, reading=8, writing, open=15 # values are 0, 8, 9, 15
valset DoorStats open, ajar, closed # note the apparent name conflicts
DoorStats frontDoor
frontDoor = ajar
@ vehicles.Values['truck'] # prints: 6
```

Layouts and Bit Fields

A *layout* is a description of a set of consecutive *bit fields* in a binary value. It describes the parts of an unsigned integer, up to 64 bits in size. The size of the layout must be a valid size of a **uint** item.

A layout is an object type with restrictions. It defines a sequence of bit fields, named or unnamed, laid out left to right, with optional values for each field. It can contain valset declarations which provide named values usable in the fields.

The bit fields in a layout are public and can be set or retrieved.

The form of a layout declaration is:

```
{ layout layout-type-name [size]
  [valset declarations]
  [field declarations]
}
```

The *size* is 8, 16, 32, 64. If not specified, a size is chosen.

The field declarations define bit field sizes, from left to right. The bit position is not identified.

A field declaration can have one of these forms:

- an unnamed field of *nn* bits size, default *value* is zero-valued bits:

  nn [*value*]

- a named field of *nn* bits size with an integer or bool value:

  nn [*type*] name [*value*]
  where *type* can be **uint**, **int**, or **bool**; **uint** is default type. Size *nn* for **bool** must be 1.
  A 1-bit field cannot be type **int**.

- a named field of *nn* bits size with a valset value:

  nn valset-type name [*value*]
  where the default is a valset value and *nn* must be suitable for the valset value.

A sequence of fields of the same size can define several fields. Example:

5 uint R1,R2,R3
4 int x1, uint x2

Missing bits on the right are unnamed and assumed to be zero-valued.

A layout name can be used like a type cast on a **uint**, **int** (any size) or **float**. For example:

```
int signedVal = -2
{ layout chkSign
  1 bool Neg
}
if signedVal.chkSign.Neg then @ "value is negative."
```

Also, a variable or array or procedure can have a layout name as a type. In this case, a size is required.

Layout Constants

A layout constant has a form resembling an array constant. It is a layout type name followed by a square bracket-enclosed set of field names and values. The layout name can be omitted in initial values if it is discernible.

Example of a simple 8-bit layout and a constant definition:

```
{ layout control 8
  1 bool start=false
  3 # unnamed zero bits
  4 secs
}
control starting =[start=true, secs=6] # defines bits 1_000_0110B8
```

Strings

A string is an object. Strings are immutable. The bytes in a string are numbered from 0.

A string's length is limited to a positive value that fits in an **int** type variable, about 2 billion bytes.

Strings are not fixed length. There is no separate "character" or "char" type.

Like arrays and defined objects and structures, strings are a reference type. String values are immutable.

A string of length 0 (a constant such as '' or "" (a pair of apostrophes or quotation marks) has **Count** value of **0** and it does not test as **nil**. A string which has no assigned value tests as **nil** and also has a **Count** value of **0**.

The characters of a string can be indexed using an array subscript notation. The first character has index 0. A negative index counts from the end; -1 means the last character.

A string *slice* is expressed in the form abc[*start***:***end*] where *start* and *end* can be positive or negative, and abc is a string variable, not an expression. Omitting *start* in a slice is the same as specifying 0, and omitting *end* is the same as indicating the last character or -1.

A *substring* is extracted using the form abc[*start* **for** *num*] where *start* can be positive or negative, *num* is positive and nonzero, and abc is a string variable, not an expression. The *start* position is the first character, same rules as a slice, and cannot be omitted. The length of the result is *num*.

Invalid *start* or *end* or *num* yields a zero-length string.

Examples:

```
string st = "abcdefghijk", xyz
xyz = st[0]        # sets xyz to string 'a'
xyz = st[-1]       # sets xyz to string 'k'
xyz = st[2 for 4]  # sets xyz to string 'cdef'
xyz = st[-3:]      # sets xyz to string 'ijk'
xyz = st[:4]       # sets xyz to string 'abcde'
xyz = st[-6 for 5] # sets xyz to string 'fghij'
```

Strings are implemented with structure **@String**.

String Constants

String constants are enclosed in apostrophes (') or in quotation marks (") or preceded by a grave accent. The difference is that a string constant enclosed in quotation marks permits escaped character sequences as listed in the table below and it also allows simple insertions from any expression that can be typecast to string. The insertion is denoted by enclosing the expression in curly braces.

If a string constant begins with an apostrophe, an embedded apostrophe must be preceded by a reverse slash. Similarly a quotation mark embedded inside a string constant which starts with a quotation mark must have a preceding reverse slash.

A string constant which has no embedded blanks or insertions or special codes can be written with a prefixed grave accent. It does not require a closing grave accent. It may not contain a grave accent. These are called simple string constants.

A simple string constant ends when a blank, a comma, a right parenthesis, a comment mark, or the end-of-line is encountered. It may not be zero length.

Here are some examples:

```
`A        `!        `Status        `*        `User-ID
```

Table: Escaped Characters in a String Constant

Escape Chars:	Meaning:
\n	End of line
\r	Carriage return
\f	Line feed character
\t	Tab character
\	Reverse slash
\{	Left curly brace
\dd	Hexadecimal value dd, 2 hex digits **dd**

There is no escaped character support for the BELL or VERTICAL TAB or FORM FEED codes or other holdovers from teletype days.

There are no decimal or octal or binary character codes in a string; only hexadecimal.

A reverse slash before any other character is retained as a character in the string.

These insertions are recognized only when enclosed in quotation marks.

String Operators

String values are concatenated with a vertical bar operator, as in:

```
a = a |'*'
a |= `*    # another way to do that
```

There are special operators for strings. The "after" operator **<:** sets s2 to the remainder of the string s1 after finding the left operand, "value":

```
s2 = "value" <: s1
```

The "before" operator **:>** sets s2 to the value in s1 up to (before) the right operand, the exclamation:

```
s2 = s1 :> "!"
```

The searched value is pointed to by the angle bracket. These can be combined to obtain a "between" expression:

```
str = 'abcdefghijklmnop'
between = 'efg' <: str :> 'mno'   # yields 'hijkl'
```

The string split operator, a slash (like divide), splits the left operand using the right operand as a separator. The result is an array of strings:

```
string ident = "dept-group-member"
string[] idparts = ident / `- # yields array: ['dept','group','member']
```

String Expressions

Concatenation example:

```
string drink = 'tea'
string tasty = 'iced '|drink # concatenated
```

The characters of a string are numbered from 0, counting bytes, not Unicode "code points".

A string can be indexed with a subscript like an array.

Assignment copies a reference. The strings point to the same value

Strings can be compared with the comparison operators. To be equal, the length and all character values are the same.

Table: Built-in String Functions

Function:	Description:
ByteValue(uint.32 n)	Value of a single byte of a string, as a **uint.8**
Caps	String, the first letter upper case, the rest lower, in each word
Copy	Returns a copy of the string, not a new reference
Count	Length of the string, the number of bytes, type **int.32**
Each	A generator function
Find(string s)	Position (int.32) of the first substring **s**, **-1** if not found
FindRight(string s)	Position (int.32) of the rightmost substring **s**, or **-1** if none
Lower	String returned with all letters lower case
Replace(string s1, s2)	String s returned with each substring **s1** replaced with **s2**
Reverse	String with characters reversed, not preserving Unicode chars
Trim	String returned with leading and trailing spaces removed
TrimLeft	String returned with leading spaces removed
TrimRight	String returned with trailing spaces removed
Upper	String returned with all letters upper case

The built-in string functions are defined in the standard library **std/string**. Note that Count is also an array function.

Caps, Lower and Upper only change the case for the twenty-six letters **A** through **Z**.

Trim functions remove spaces and also \n (LF), \r (CR) and \t (TAB) codes.

Examples:

```
string s = 'abcdefghij', ss, st
int pos = s.Find('def')    # set to 4
ss = s[:7]   # truncates to 8 characters
st = 'b'<:s:>'h'  # gets 'cdefg' substring
string txt = 'apple, peanut, cheese '
# set array words with 3 strings: 'apple', ' peanut', ' cheese '
string[] words = txt/','  # split, then need to trim words
words = words.Trim  # trim each of the words
```

Typeset Specification

A *typeset_spec* identifies a named set of types which can be used to indicate which types are supported by a procedure. It is specified with the **typeset** keyword, for example:

```
    typeset arithType = int uint float
```

The typeset name arithType above represents a type which can be **int**, **uint** or **float.** The specified types allowed must be predefined (basic) types or an asterisk, which means any object type.

A restricted size (like **uint.16**) may be specified for types **int** or **uint**.

This may be specified in an object definition.

The *typeset_name* is available as a type name in procedure and other definitions in the object or global definition. Any of the eligible types for that name will qualify for that usage.

The *base_type* on a procedure definition can specify which types are permitted by naming the *typeset_name* in the base type or as the type of a parameter or as the result type or an index type.

The standard library file **std/typesets** defines several usable typeset names.

Procedures: Functions and Methods

A *function* returns a value or reference, and thus can be used in an expression. It is invoked by naming an object on which it is defined, followed by a selector period, then the function name, then parameters, if any, in parentheses. The object is an implicit parameter to the function. A function may have no parameters and no parentheses.

A *method* does not return a value. It is invoked like a function, except that no parentheses are used when there are parameters.

Functions and methods (these forms are called *procedures* in CB) are defined in an object type definition, except when they are with no base type specified or a basic type or known valset type they can be defined outside an object.

Passing Parameters to Procedures

A procedure may be invoked upon an object. The object is then implicitly passed as a reference parameter internally referred to as **$**.

The procedure name is applied to the object by placing the procedure name after a selector period on the right of the object.

A function with no parameters, or with all parameters using defined default values, is invoked with no parentheses.

A method never uses parentheses on its parameter list, even when parameters are passed.

Declarations

A *declaration* is a non-executable statement or construct. A declaration defines the type and use of names in the language. The following are declarations:

- A type declaration defines variables, arrays, lists and functions.
- A new object type or structure definition.
- A **valset** declaration.
- A named constant declaration.
- A **typeset** declaration.
- A procedure definition.
- A constructor declaration.
- An operator declaration.
- A type cast declaration.
- A layout declaration.
- A **pragma** declaration.

Declaring Variables and Arrays

Places where declarations (other than objects and procedures) can be declared:

- In a block, also a switch case block.
- In an object definition.
- In a procedure definition.
- Variables for the index and value are implicitly declared, with limited scope, in a **for** construct. The index is the index type of the array. The value is the same type as the array.
- Variables are declared with type information before them. Examples:

```
int number
string name
bool test
```

- Arrays are declared with square brackets after the type. All members must be the same type and all indexes are uint or int or string.
- Variables and arrays can have initial values specified after the name. Examples:

```
int mnx = 23
string[] ap = ['start','stop']      # indexes are 0, 1
uint.64[string] salary = ['CEO':250_000, 'analyst':45_000]
```

- Variable names and procedure names in an object (the same name space) must be distinct. However, a variable can be the same name as a procedure which is defined in a different name space, and vice versa.
- Valset value names are in separate name spaces for each valset.

Named Constants

A named constant declaration defines a name which can be used for a constant value. This declaration can appear in or outside an object declaration, or any block. It has the following form:

[*access*] **const** *name constant_value* [**,** *name constant_value* ...]

Within an object type definition, *access* is public unless **shared** or **private** is specified as access. Access in a block is local to the block and cannot be specified.

The type of the name is the type of the constant. Non-basic (object) constants must show their type.

The defined *name* by convention is all capitals where letters are used, but this is only a recommendation. Predefined named constants begin with an underscore and an uppercase letter.

The *constant_value* is a single constant of the declared type.

Examples:

```
const MAX_TIMES 100, INTEREST 0.035
const PROMPT 'User ID? ', BETA 4.336E0
```

Named constants defined outside any object have global scope. For example, the mathematics constants pi (π) and **e** and others are defined as float values to a large number of digits by these lines (shown truncated) in a global definition in the library **std/math**:

```
const _PI 3.14159265358979323846264338327950
const _E 2.71828182845904523536028747135266
const _LN_10 2.302585092994045684017991454684
const _ONE_DIV_LN_10 0.43429448190325182765
```

Statements

A *statement* is a single line executable action, or a multi-line executable *construct*.

Statements are only allowed in a procedure, including a method, function, generator function, operator definition, type cast definition.

Statements which control the flow of execution, meaning the progression from one statement to another, are called *control statements*. These are the **if**, **for** and **switch** constructs, the block construct, subordinate statements **case**, **else, final**, **while**, **until**, **return**, **if**, **break** and **repeat**.

Other statements involve variables and arrays and procedures and expressions. These are:

- The *assignment* statement uses the assignment operator **=**.
- The *incrementation* statement is actually a simple expression using the postfix incrementation (**++**) or decrementation (**--**) operator. The side effect change makes this also a statement. It is considered an assignment when it stands alone.
- An array assignment.
- A method call is a statement.
- A function invocation can be a statement if it stands alone, not in an assignment or expression. The result is ignored.
- A construct is considered to be an executable statement.

If Statement

An **if** statement has the form:

```
if bool_expression statement
```

where *bool_expression* is a scalar expression that evaluates to **true** or **false**, and *statement* is a single executable statement, not **if** and not a construct. The statement can be an assignment or incrementation or a method call, which must be preceded by **then**, or it is one of the statements **break**, **repeat** or **return**. It cannot be a block, **if**, **while**, or **until**.

If Construct

The *if_construct* has the following form:

```
{ if bool_expression
        statements
[else if bool_expression
        statements]...
[else
        statements]
}
```

A *bool_expression* is a scalar expression that evaluates to **true** or **false**. There is no statement or keyword **then** after the *bool_expression*.

Keyword **else** is not a statement, but begins a section of the construct.

Additional conditions (**else if**) are allowed after the first one, and before optional **else**. If the first **if** condition fails, it tries the next, etc., then the **else** section is executed if all fail. These have the word **else** before **if**. Only one of the conditions is executed, the first to be true.

Note that the similar-looking **if** statement is considered a single statement, not part of a construct.

Switch Construct

The *switch construct* provides a selection of alternative actions, somewhat like a conditional block.

It has the following form:

```
{ switch expression
case constant [,...]
      statements
[case constant [,...]
      statements]...
[else
      statements]
}
```

The expression on the **switch** must be an **integer** or **string** or **valset** expression. It shall not be an array. The constants on the **case** lines must all be the same type as the switch value. String constants must not have value inserts.

The expression in a switch statement is evaluated once, then execution proceeds to the **case** or **else** which matches the value of the expression. The following statements (the *case_block*) are executed. Execution goes to the ending curly brace if it falls through the statem**e**nts into a **case** or **else**, rather than fall through into the following case block.

The **break** statement can be used to override the flow of execution. A **break** sends control to the ending curly brace.

No statements are allowed between **switch** and the first **case** or **else**. There must be as least one case block in the construct, not just **else**. Statements which are not allowed in a switch block or case block except inside a contained looping block: **until** and **while**.

An empty block after a **case** or **else** does nothing. It immediately exits the **switch**.

The set of constants must not have any overlapping values.

A **case** line can list multiple values separated by commas.

The **else** block (which is like a case block) indicates selection of any other unspecified value. It must follow all **case** sections.

Example of a switch:

```
int mv = 8, xlv
{ switch mv
case 1, 5
  xlv = 0
case 6,25,50,59
  xlv = 1
else
  xlv = -1
}
```

Block Construct

The *block_construct* is a block which does not loop. It does not contain a **final** block, and there are no looping control flow statements (**repeat** or **while** or **until** or **do** or **for)** except **break**, which may be conditional. It has the form:

```
{
  statements
}
```

A block with no loop:

```
{ int jxx = 1
  statements
}
```

A non-looping block may have declarations which are local in scope. A variable or array declaration may be on the same line as the initial curly brace.

Do Construct

A block with a **do** statement on the first line is a looping block. It may have a **final** block.

```
{ do [initializations] [while_until_statement]
   statements in the block
[final
   statements]
}
```

The initializations following **do** are executed once only. Any variables or arrays defined in the initializations have scope over the entire construct. Control flow goes back to the line after **do** from the end of the final block (or right curly brace) or to the while or until specified on the **do**. Initializations follow the rules for declarations.

The statements after **final** are executed every time through the loop. A **repeat** sends control to the final block. This is where an incrementation is placed.

Example of a loop:

```
{ do int k = 1024 until k =? 0
   # statements here
final
   k = k/2
}
```

This loop assigns a value to **k** once, then repeats the statements in the block, dividing **k** by 2 at the end, and goes back to **until** after the **do**. Since type **int** for **k** is specified, the variable **k** has local scope in the construct. The loop terminates when **k** is zero.

A simple loop example:

```
{  do while test_expression
   statements
}
```

If *a* variable is not typed in the initializations and it is assigned a value, it must be a valid existing typed variable, and the value is retained after the end of the construct for its normal scope.

If it is declared with a type it has that type and has scope local to the construct and the initialization is done once at that point. The value is not retained outside the block.

While or **until** may be placed after initializations on a **do** or anywhere after that as a statement in the **do** or **for** construct before the **final** block.

While

The **while** statement has the form: **while** *test_expression*

The *test_expression*, a scalar expression of type **bool**, is evaluated and control does a **break** to the **final** block or the ending curly brace if the expression is **false**. The presence of a **while** makes the block a loop. The **while** statement is equivalent to:

if not (*test_expression***) break**

Until

The **until** statement has the form: **until** *test_expression*

The *test_expression*, a scalar expression of type **bool**, is evaluated and control does a **break** to the **final** block or the ending curly brace if the expression is **true**. The presence of an **until** makes the block a loop. The **until** statement is equivalent to:

if (*test_expression***) break**

These statements are valid only in a **do** or **for** construct and an inline **for**.

For Construct

The **for** construct loops through an array or string or any object based on a traversable structure, handling each value. An array can be used:

```
[1]    { for valu[[k]] from array_expression
             block
       [final
             block]
       }
```

The expression is evaluated once. Then each element of the array is extracted, assigning the (local scope if not already typed) variable *val* the value, repeating the *block*. The type of *valu* defaults to the same as the array type. The type of *k* is defined by the array. Only the array elements which exist (the index exists) are used. Values are presented in index order, with or without the index **k**.

Example of [1]:

```
       float[string] cost = ['notebook':2.55, 'pen':0.77, 'paper':1.98]
       { for cost[item] from prices
         @ "item={item}, cost={cost}"
       }
```

A string can be used also, similar to an array. Variable *valu* will be assigned each character, left to right, as a single character string. Unicode characters are not recognized.

```
[2]    { for valu [[k]] from string_expression
             block
       [final
             block]
       }
```

The expression is evaluated once. Variable *valu* will be assigned each byte, left to right. The local scope variable **k** is assigned the position of the byte and implicitly local (or previously declared) variable *valu* gets the value, as a type **string**, repeating the *block*. The assumed type of optional variable **k** is **uint.64**, and **k** will begin with 0. Variable **k** has local scope to the block or it may be predeclared. The string expression is evaluated once, and cannot be altered in the block.

A traversable object can be used:

```
[3]    { for valu from _Traversable_object
             block
       [final
             block]
       }
```

Any object *obj* which is an array or has a shape based on the abstract structure **@Traversable** can be used in a **for** construct. The **for** construct internally expands to initialize with the object, then increments (or iterates) using Next, and it stops when no Next value is found.

For any traversable object, such as an array, the construct:

```
{ for item from obj # the type of obj is type
   block
}
```

internally expands into:

```
{ do type item = obj.First while item.isNotNil # initializes once
   block
final
  item = item.Next
}
```

To make this statement go through a list, for example, the initial obj is the first item of the list. The variable item is a name of local scope, same type as obj.

Also, a generator function can be used.

```
[4]    { for valu from object.generator_function
             block
       [final
             block]
       }
```

This repeatedly calls the *generator_function* returning a value until it returns **nil** or falls to the function end, which causes a break.

Final Block

A *final_block* can be used in any looping construct or **do** or **for** construct.

The keyword **final** marks the start of a *final_block*. The purpose of the *final_block* is to allow actions at the end of each iteration of a looping construct, such as incrementation.

Immediately following the *block* in the looping construct, a **final** statement introduces a *final_block*, the (non-empty) block of statements after keyword **final**. This block of statements remains part of the block for name scoping. Execution flow falls into this *final_block*, through the **final** statement. Any incrementation or other action may be placed in the *final_block* if it is to be done at the end of each loop.

The *final_block* is executed on every repetition of the loop**.**

An empty *final_block* can be omitted.

A **repeat** statement in the block before **final** transfers to the *final_block*. A **repeat** is invalid in the *final_block.*

A **break** statement anywhere in the construct exits the loop.

Repeat Statement

The **repeat** statement is valid only inside a looping block or **do** or **for** construct, and is not valid in the **final** block portion. It causes execution to skip over statements to the **final** block or to advance to the next array element. In other words, to the incrementation, the **final** block if it exists, or the next containing right brace at the block end on the nearest containing loop block or **for** construct.

Break Statement

The **break** statement is used to exit the nearest containing loop, **do** or **for** construct or **switch** block. It is typically used to exit a loop when a condition is reached. It is not used for a procedure body exit.

Return Statement

The keyword **return** is valid in an object definition. It can be used in a procedure body as a statement.

In a method, it returns control to the point after the method call.

In a method, the return statement has no parameter; falling into the end of the definition causes an implicit **return**.

In a function definition, the return statement has the form:

> **return** *expression*

where the expression is the returned value of the function.

In a function, the last statement in the function must be a return with a value.

A generator function which has control falling into the end of the block implicitly returns nothing and ends the generator. Also, it ends when a statement **return nil** is encountered.

Control Flow Limitation

No executable statement may appear immediately following an unconditional control statement which breaks the normal flow of control.

In other words, a statement is not permitted after **return**, **break**, or **repeat**.

This rule does not apply to the **return** statement in a generator function, nor to the above statements on an **if** statement.

Blocks

A block acts an executable statement. The block can contain declarations and statements, intermixed. The statements and declarations are performed in order, from the top, unless a control statement changes the order of execution.

The names defined in a block have scope from the definition to the end of the block. If **private** access is specified, the scope does not extend to inner blocks.

Blocks are:

- The *for construct* or *block construct* includes the entire construct and the block executed. It also includes the *final_block*.
- The *case_block* is all of the statements and declarations from case or else in a switch until case or else or the end of the switch.
- The *switch_block* is all of a switch construct. It includes the expression on the switch statement plus all of the enclosed case or else blocks. Declarations have scope in the switch block where they appear.
- The *conditional_block* is all the statements in the braces.
- The *procedure_block* is all statements and declarations in a procedure definition. This begins at the start of the prototype portion and ends at the procedure end. This applies also for properties, operator and type cast definitions and for a constructor.
- The *final_block* is the statements after **final,** in the following block. It is not a true block because it is part of the same scope as the construct.

Defining Objects

An *object* is a named entity which has values, procedures and other functions within it. Objects are used to encapsulate data and functions, hiding details.

The object name becomes a type, an *object_type*, which is used to declare variables and arrays and procedures, etc. An *object_type* is analogous to a '*class*' in many other languages.

An object can inherit another, gaining its functions and data. Multiple inheritances are disallowed except for inheriting abstract objects, indicated by a question mark for the *object_option*.

Object definitions cannot be nested or internal.

A new *object_type* is defined in the following manner:

```
{ object [object_option] object_type [uses [object_type] [abstract_object...]]
    [typeset declarations]
    [valset declarations]
    [layout definitions]
    [variable and array declarations]
    [procedure definitions]
    [operator definitions]
    [type cast definitions]
    [prototype definitions]
    [constructor definitions]
    [pragma declarations]
    [static area, values and procedures]
}
```

The inherited *object_type* can also be one of the basic types (int, uint, bool, string, float) or a valset type. This becomes an *enhanced_type* object definition.

The object_option **final** implies the object cannot be inherited.

Abstract Objects

When *object_option* is **abstract** the object is abstract. An abstract object has no data, and no definitions for procedures, operators, type casts or properties. It can inherit only other abstract objects. It can contain the following:

```
    [typeset declarations]
    [valset declarations]
    [layout definitions]
    [constant definitions]
    [procedure prototypes]
    [operator prototypes]
    [type cast prototypes]
```

Defining Structures

A *structure* defines a special form of an object which has one anchor storage area and any number of linked leaves, or leaf storage areas. It is used to define a *shape*. A *structure_id* defines the storage of the whole structure and its parts, or storage areas. A structure must define anchor and leaf areas, in that order. No other data is defined in a structure except in an optional static area.

A *structure_id* is not an *object_type*, but it can have a static area.

Structure definitions cannot be nested or internal.

A new *structure_id* is defined in the following manner:

```
{ struct [abstract] structure_id [[index_spec]] [uses [structure_id][abstract_structure...]]
    [typeset declarations]
    [valset declarations]
    [layout definitions]
    [constant definitions]
    [pragma declarations]
    [static area, values and procedures]
    [anchor area, values and procedures]
    [leaf area, values and procedures]
}
```

The *index_spec* specifies that the *structure_id* defines an array or a matrix. It can take one of these forms:

```
[index_type]           - - - defines an associated index array
[uint var]             - - - defines a one-dimensioned matrix (row, vector)
[uint var1, uint var2] - - - defines a two-dimensioned matrix
```

The inherited *structure_id* can only be a structure definition or abstract structure. Items in each storage area are added to the same storage area.

Abstract Structures

An abstract structure is marked the option **abstract** after the keyword **struct**. It is like an abstract object except all prototypes are contained in **anchor** or **leaf** storage areas.

Names in an Object or Structure

Variables and arrays and objects are declared in the form:

[*access*] *type_name*[*shape*] name [**=** initial_value], ...

Additional names, separated by commas, can follow when the *type_name* and *shape* are the same, omitting the type and shape.

Access: Visibility of Names

Access is one of the keywords **private** or **shared**. If unspecified, names are public. It controls the places where a name of a variable, array, procedure, function or an object are valid, or visible.

Private

The keyword **private** means visible in the current object only. It means the name cannot be seen in inheriting objects.

Shared

A shared name is visible in the object and in objects which inherit the object, but is not visible outside the objects. The name is not visible from inherited objects. It is not public. This is like the keyword "protected" in some other languages.

Public

When the access is not specified, the item is public and is visible across the object and any inheriting objects and the member is visible as a member of the object.

Variables, procedures, arrays and structures in an object are public if no access is specified. There is no reserved word (keyword) "public."

Storage Areas - Static or Anchor or Leaf

Within an object definition, variables, arrays, etc. of the object are normally allocated in each instance of the object. Also, procedures in the object apply to and use these instance values. Definitions of typesets and valuesets and constants apply to all parts of an object.

Within a structure object, all public data items must be in a storage area.

Procedures across all areas (static, anchor, leaf) of a structure are defined across the structure, before the anchor area. No duplicates except overloaded procedures are allowed. Procedures are defined outside of the storage areas. The anchor or leaf areas are like unnamed objects.

Static Area

Variables, arrays and procedures (function, method or get/put property) in an object can be defined in a **static** area. All static names are public; no *proc_option* is allowed.

This implies that there is a single copy accessible under the type name, not in an instance of the object, but tied to the object type.

The static area in an object type is defined by this form:

```
{ static
  # variables, arrays, procedures in the static area
}
```

These static items are not accessible as members of a variable. Access is through the object type name as a base. For example, static value **val** in object type **ABC** is referenced as:

ABC.val

Static values do not go away when objects are deleted or go out of scope. Static procedures do not have an instance. They can only use static values.

A static area can be defined in structure objects or non-structure objects.

An object (not a structure) which has all data and procedures only in a static area is called a static object, and it implies the **final** option. A static object cannot be used as a type, and cannot define operators or constructors or type casts. Its contents are referred to based on the object type name.

Anchor Area

The anchor area in an object definition is allowed only in a structure object type. Items in the anchor area are shared as a single copy per structure variable or value, accessible across linked leaves in the structure. These items describe the entire structure, and define a link to the first or root leaf, possibly a count of members, and other items needed.

The form of the anchor storage area is:

```
{ anchor
  # variables, arrays, procedures in the anchor area
}
```

When a structure is created by a constructor, using the object type (structure) name, only the anchor data is allocated. When members are added or deleted, they contain only the non-anchor (node or leaf) values, and are found using the anchor. The anchor area values may be updated, depending on their access.

Procedures in the anchor area affect the anchor content and are invoked with the structure name as a base. The anchor contains at least one leaf reference name.

Leaf Area

The leaf area values are those which are in a structure, but not static and not in the anchor area. The "leaves" are members or nodes of the structure.

The word **leaf** means a reference to a leaf, as if it is a type. It is also a constructor, referenced as the object type, a period, and the word **leaf**.

The leaf area is defined similarly to the anchor area:

```
{ leaf
  # variables, arrays in the leaf area
}
```

New members (leaves) are allocated in expressions via a constructor. This adds only the space needed for the non-anchor (leaf) data. These values may include a root or parent link, left or right links as needed, an index if there is one, and the node value. For efficiency, these items may be rearranged by the compiler.

A leaf constructor definition begins with the words **new leaf**.

The word **leaf** can be used as a type name inside the leaf area.

Procedures in the leaf area affect the leaf referenced by the link in the anchor area or a named leaf variable.

The same procedure name can be defined in both anchor and leaf areas. They are different procedures.

Structure Example

The following defines a simple structure called a Ring. This structure has no index, and no first or last member. It is not traversable in a **for** construct because it has no first member.

```
{ struct Ring
 { anchor
   leaf Current # points to a leaf
   uint Count
   { method Clear
     Count = 0
     Current = nil
   }
 }
 { leaf
   shared leaf fwd, back
   _DataType Data
   { func Next gives leaf
     if Count < 2 return Current
     Current = Current.fwd
     return Current
   }
   { func Prev gives leaf
     if Count < 2 return Current
     Current = Current.back
     return Current
   }
   { method Clear
     if Current.isNil return
     Count--
     { if Count.isNonZero # still has at least one
       Current.back.fwd = Current.fwd
       Current = fwd # advance to old fwd
     else
       Current = nil
     }
   }
   { method Insert(dataType nData)
     leaf newLeaf = Ring.leaf # allocate one leaf
     newLeaf.Data = nData
     { if Count.isZero # empty ring
       newLeaf.fwd =  newLeaf
       newLeaf.back =  newLeaf
     else
       newLeaf.fwd = Current
       newLeaf.back = Current.back
       Current,back.fwd =  newLeaf
       Current.back =  newLeaf
     }
     Current =  newLeaf
     Count++
   }
 }
} # end structure Ring
```

Object Values and Constants

Object values are constructor calls with parameters and values.

If the values assigned are constants, the result is an object constant.

The object type name is used, followed by a set of name=value assignments in a pair of parentheses. All value assignments with a default value are optional and may appear in any order. The names are defined by the constructor or defaulted to public variables and put properties. Example:

```
{ object Animal
  string kind, says
}

Animal Tabby = Animal(kind='cat', says='Meow')
```

Global Definitions (Outside an Object)

Some definitions can be defined outside an object type definition. Object types must not be nested. Definitions outside an object are called global definitions.

These definitions are allowed outside an object definition:

- `object type declarations`
- `typeset declarations`
- `layout type definitions`
- `named constants`
- `valset type definitions`
- `methods with no base type specified`
- `functions and methods on a basic type, defined object type, a valset type or a typeset`

The names defined are global names, as if the global definition is inherited by every object. Local definitions override. There are no global variables or arrays.

Methods defined which are not defined on a type have no base type. These methods are not based upon any object. They shall not refer to $. A method with no base type cannot be defined inside an object.

Procedures are allowed to access defined constants, and they are allowed to use basic type or known or global type definitions.

Procedures defined outside an object but based on that object can only use public names and properties in the object.

Inheritance

When an object type inherits another, it gains access to the public and shared variables, properties and procedure definitions.

An object type definition can name a single parent object type to be inherited.

A variable *V* of a given object type *T* fits the words: *V* "is a" *T*.

If *T* inherits *TT*, then *V* "is a" *TT* also.

As an example, with V1 of type CAR and V2 of type SUV, and each of these types inherit a type AUTOMOBILE, then we can say both V1 is type AUTOMOBILE and V2 is type AUTOMOBILE. This allows a procedure to process either V1 or V2 as an AUTOMOBILE.

Every object (all things are considered "objects") implicitly supports this built-in function to test or inquire about the object's type:

```
    isType       - - function, returns the type name as a string.
```

Instantiation of an Object

An object variable is a place-holder for a reference to an object. It is instantiated when it is assigned a reference to an instantiated object value or object constant. Creating an object constant using the object type also creates a value and instantiates the object. Assigning or initializing a value inside the object also instantiates it.

The static components are instantiated when the object is declared.

Default Value at Instantiation

An integer or float variable has a default value of 0.

A string has a 0-length string and the value **nil** as default.

A bool variable defaults to **false**.

A **valset** variable defaults to a value of 0 even if no name is associated with 0. A value with no associated name cannot be compared.

An object or structure has a default value of **nil**.

Enhanced Type Objects

The inherited object type can be one of the basic types (int, uint, bool, string). This becomes an *enhanced_type* object definition.

The enhanced object is restricted in several ways:

- It can contain only one non-static public data member, of the same type as is inherited.
- It can contain static members, also constants.
- It can contain functions and procedures.

Like any inherited type, the new type is a type that is inherited. For example, a new type Degrees which inherits float is also a float.

The enhanced object type is a way to specialize values, One enhanced type cannot be mixed with another even though they both inherit the same type. For example, a Meters type and an Inches type can inherit float, but they cannot be mixed by mistake. Functions may be provided for conversion. Automatic type casting does not happen.

An enhanced type is permitted to be used as a value of the inherited type. This allows mathematical functions to be applied. However, if an expression mixes types, the mix is restricted. Addition and subtraction must use the same type, and the result carries the type, so inches plus inches is still inches.

However, multiplication and division are allowed only with the inherited type. This allows "scaling" or "discounting" to happen.

An enhanced type cannot be cast as its inherited type, losing the enhancement. This protects from changing "units" like trying to type cast a metric Length to inches by using **Length.float.inches**. The correct way is to provide functions or procedures which convert directly.

An object which inherits an enhanced object type is not restricted. It may contain other data.

Properties

A *property* is a named object member which resembles a variable.

There are two kinds of property, get and put. A *get* property obtains a value; it is a readable value. A *put* property stores, reads, replaces, writes or sets a value. It is used as an Lvalue. A get property and a put property can share a name, which makes it appear like a normal public variable.

A put property name cannot be used as a base value with a selector following because it returns nothing.

When a property name is passed as a value to a procedure in a parameter, it is treated as an expression value, using the get property.

A property name in an object has public or shared access, not private.

A property name can be the same name as a parameter name in a constructor.

A property cannot be overridden.

A property has no parameters.

A put property paired with a get property of the same name allows both read and write actions. Without a get property of the same name, a put property acts as a write-only variable.

A property that is both get and put can be used with unary increment or decrement operators.

Defining a Property

A property name is defined by a construct like this:

```
{ property type_name name [ return expression ]
[
put
   action_block
]
}
```

There are no parameters, no access, no shape.

An optional *get_property* is defined by the **return** and *expression* on the first line. If there is no **put**, the curly braces can be omitted for a short form get property.

A *put_property* is defined by **put** and the *action_block*. The first line of an *action_block* can appear on the **put** line unless it is a block or construct beginning with a curly brace..

The *name* must appear as a value in an *action_block*. The *name* is evaluated once.

The *action_block* is usually an assignment statement to a local non-public variable, with the *put_name* referenced on the right side expression, or it is a method call in the object, using the *put_name* as a parameter. Additional statements are allowed; a **return** statement is not allowed.

A *put_property* is used when the *put_name* appears on the left of an assignment.

Property Example

Define an object type Rect, with length and width, storing these in millimeters but showing external values in inches.

```
{ object Rect
  const MM 25.4 # millimeters per inch
  private float len, wid # internal values are saved in millimeters

  # "read" (get) properties - - external is in inches
  func Length gives float len / MM
  func Width gives float wid / MM
  func Area gives float Len * Width # using the properties

  # "write" (put) properties - - external is in inches
  func float: Length len = Length * MM
  func float: Width wid = Width * MM
}
```

Defining Procedures

Procedures are defined inside an object definition. Procedure definitions cannot be nested. They can also be based on a basic type.

A procedure can be defined on an array of any type by specifying an object type followed by a pair of square brackets as a shape. The array index type must be specified to apply to a specific type of index. The array type must have defined indexing.

For all procedure definitions, the *base_type* and its following colon can be omitted if it is the object type currently being defined.

A procedure definition generally follows this form:

```
    [return] proc_kind base_type shape: proc_name (params) gives ret_type shape
```

where *proc_kind* is **func**, **method**, or **new.** Some parts may be omitted in some definitions.

The *shape* is used when the base is not scalar. It describes an array or structure.

Identifying the Base Type

In a procedure definition, the *base_type* is a type name followed immediately by a colon.

Inside an object type definition, the current object type is assumed as the base type for any name defined in the object or inherited. The base type can be a basic type name or inherited type name or omitted.

The base type name can be the name of another object type.

Procedure Overloading and Overriding

Within an object definition, procedures can have the same name, but differ in their parameter specifications. One function **Fun** may have an **int** parameter while another **Fun** has a float parameter. This is called overloading.

An object may inherit another object, and both define the same procedure. The inheriting object is said to override the inherited procedure. If a procedure or property is a prototype, there must be an overriding definition in an inheriting object.

Procedure Options

The *proc_option* on a procedure definition is an optional access word (**private**, **shared**) and any of the keywords **repeat** or **final**. These words can appear in any order. Specifying **repeat** is allowed only on a function, implying a generator function.

A procedure with the *proc_option* **final** or defined in the **static** area cannot be overridden. This can appear with **shared** or (the default) public, but not with **private**.

A prototype describes a procedure which must be defined in an inheriting object definition. The presence of a prototype implies that the current object definition requires a definition.

Methods

A *method* is a procedure with no returned value. It is used as a statement, and is not allowed in an expression. The return type specification is not specified. The parameters are not enclosed in parentheses. The form for a method definition is:

```
{ [proc_option] method [base_type[shape]:] name [params_spec,...]
            block
}
```

A method with no base_type cannot be defined in an object. In an object, an omitted base_type defaults to the object type.

There is no short form.

Functions

A *function* is a procedure which returns a value, to be used in an expression. The form for a function definition is:

```
{[proc_option] func [base_type[shape]:] name [(params_spec,...)] gives ret_type[shape]
     block
}
```

The *proc_option* keyword **repeat** implies it is a *generator_function*. Keyword **final** prevents the function from being overridden.

An omitted *base_type* (and colon) in an object definition defaults to the object type.

The *shape* is used with a type. It describes an array or structure or both.

The *params_spec* (including parentheses) is omitted if there are no parameters.

The *ret_type* is the function's returned type. It must be specified, even if it is the *base_type*.

The *ret_value* is an optional returned value expression. If the returned value is shown, no function block is used and the surrounding curly braces are omitted. This simple form of a parameterless function is defined in a single line:

```
[proc_option] func [base_type[shape]:] name gives ret_type[shape] ret_value
```

Constructors

A *constructor* is optionally declared in an object type definition. The constructor is called when the object is instantiated, or when the object is initialized. A constructor appears in the object type definition and has this procedure definition format:

```
{ new [(params_spec, ...)]
      block
}
```

A constructor is public. There is no *base_type* and the implicit function name *type_name* is the name of the object type when the constructor is invoked as a function using the *type_name* as a way to create a new object.

The parameter names may be the same as public names, or they must appear in the constructor's block as references in an expression.

If there are no parameters on a constructor it will be invoked by default when an object is instantiated. If a constructor is not defined, default instantiation occurs.

A constructor with no parameters is called by the object type name alone.

Constructors with different parameters are overloaded.

The statements can use temporary variables, etc., and they can access the items in the object.

One purpose for a constructor is to set static values and private members and perhaps open a database or file.

A constructor must have statements in the block. The procedure block cannot be empty.

A constructor can be called with the "name=" omitted for each parameter. The values must be in the order defined for the parameters.

Simple Constructor

A shortened and simplified constructor definition has the form:

```
new (variable=value,...)
```

or:

```
new
```

It has no block. It assigns public variables or put property names with the values as shown in the parameter list.

Default Constructor

The constructor does not have to be defined. A supplied default constructor sets public values by naming them in the parameter list with a default value. The default one is provided, with parameters which are the names of the public variables. If all public names are to be given default values, the default constructor is simply the type name.

Defining Operators

Most of the operators can be defined for object types, in the object definition. An operator retains its precedence and its associativity. The bool operators **and**, **or**, **not** cannot be defined or redefined.

The assignment operator (=) cannot be redefined for the basic arithmetic or bool types. For objects other than structures, it defaults to copying all data. For structures, it defaults to copying the anchor portion only. The default structure for array defines assignment to set the current position to the lowest key value.

The binary operators **+**, *****, **-**, **/**, **|**, **&** and **~** and the comparison operators can be defined by creating a function with a base type as needed, a single parameter, and result type shown, and in place of the function name the operator *op* is shown:

```
{ func [base_type[shape]:] op [type[shape] b] gives ret_type[shape]
        block
}
```

where *type* is usually the same in all three places. The *base_type* can be omitted, implying the current object type. Types that are the same as previous can be omitted.

A simple definition can be in one line, as shown for addition of an **int** to an object:

```
Obj: func + int y gives Obj valu + y # valu is an int member in the object
```

Equality (=?) comparison is already defined to mean member-wise equality for objects, but can be redefined for any named object type. It cannot be redefined for basic types.

If there is no parameter "*type* b" then *op* can be ++ or - -, defining *op* as a postfix operator, or *op* can be + or - or ~, defining these as unary prefix operators.

It is not necessary to define subtraction if both addition and unary minus are defined. Similarly, if =? and > are defined, all other comparison operators can be derived. Addition and multiplication are commutative operators; if two types are used the opposite pair is inferred.

Operators cannot be redefined for the basic types bool, int, float or string. Defining an operator does not change its precedence or its associativity. Invalid operations on basic types cannot be defined. Postfix ++, for example, cannot be defined on bool or string.

Postfix or unary operators cannot be defined as binary operators and binary operators cannot be defined as unary or postfix.

The returned value is a reference for a defined object type, and a value for a basic type. The *shape* is used when the base is not scalar. It describes an array or matrix or structure.

Defining Type Casts

Similar to the definition of an operator, a type cast can be defined in an object. The following is the form for defining how to evaluate **abc.***ret_type*, where **abc** has the type *base_type*:

```
{ func [base_type:] gives ret_type
      block
}
```

Or, the simple form:

```
func base_type: gives ret_type return_value
```

Note that no function name or operator is shown. The *ret_type* is used like a function with no parameters.

If the returned value is shown on the first line no block is used and the left brace is omitted.

The base_type is the type ending with a colon, the ret_type is the type preceded by a colon. These types must be different.

The *base_type* can be a basic type so long as the *ret_type* is one which has no predefined definition for the type cast, except you can define type casting to a string.

An array or a structure can be type cast if the data members can be the new type. The *shape* is used when the base is not scalar. It describes an array or structure.

Example: a simple definition, in one line:

```
func Obj: gives string valu.string
```

Static Procedures

A procedure may be in the **static** area or in a static object, meaning it is invoked only by referencing the object type name as a base. It shall not use **$** in the body since there is no instance. It may only use or modify static variables. A static procedure cannot be overridden (it implies **final**).

Procedure Prototypes

An inherited abstract procedure must be defined in the inheriting object type. The abstract procedure remains abstract in inheriting objects if not defined, carrying the required definition to the next level.

The Return Specification

The *return_value* is not allowed for a method.

The *return_value* is an expression value to return. It may use constants and other values known in the object or parameters of the function. Variables in the instance are accessible.

If there is a *return_value* expression, the function block must be empty, and the function returns a value which is the expression.

The *ret_type* or *param_type* or *base_type* can be a basic type such as **int**, **uint**, **float**, **bool**, **string** or it can be an object type name, and it can be an array specification (with type and index type), a **valset** type name, or a **typeset** name.

When *base_type* is an object type, the procedure has access to shared or public internal data and properties and procedures inside the definition and data shared from an inherited object type. Private data is hidden unless the procedure definition is inside the object.

The base object is implicitly passed by reference, regardless of type.

Basic Object Type Procedures

A *basic_object_type* procedure is not based on a defined object type or valset type. The base_type can be a basic type (**int**, **float**, etc.).

A procedure can be defined with no base type. It has no object members or values.

The object value can be referenced in the procedure block as **$**.

Parameters Specification

The parameters specification *params_spec* defines the expected parameters for a procedure.

The *params_spec* is omitted when there are no formal parameters, or it is:

```
    param_type[shape] param_name [= constant_expression]
```

for each parameter. Parameters are separated by commas. The parameter type *param_type* can be omitted if it is the same as the previous *param_type*. The first parameter type can be omitted if it is the same as a function or operator type.

The *shape* is used when the base is not scalar. It describes an array or structure or a combination.

An array or string or stack or valset type or named object or structure type by default is passed by reference. A reference item can use the function **Copy** on an actual parameter to force a reference item to make a copy then pass a reference to the copy.

All value variables (types int, uint, float, bool) are passed by copy. An array element passes the value by copy. The array is not changed when a copy is passed.

A property always passes a get value as an expression, and a put property cannot be passed.

A variable passed by copy delivers a temporary copy to the procedure. If the procedure alters the copy, the original is preserved.

An expression always passes a copy. An item enclosed in parentheses is a simple expression. An array value or constant passes a copy.

A parameter declared to have an object type implies the actual parameter is an object of that type or it inherits that type. Types are not automatically cast, but a shorter int or uint will automatically widen.

Parameters which are variables may have a default value expressed as a constant expression in the procedure definition. Any such parameter can be omitted in a reference if it has no following explicit actual parameter which has no default value. The value passed is the value of the constant expression.

If all parameters have default values, an invocation can name any of the parameters in any order, specifying the name, an equal sign, and the value. This applies also to constructors.

Access

Access is the degree of encapsulation, or the accessibility of names in an object.

The keyword **private** is allowed as *access* on a procedure definition inside an object or structure definition. It means it will not be visible in inherited types.

Property items are like variables in appearance, but may in fact be implemented as a procedure, or they may be implemented as a public variable if the compiler finds this is safe. They are possibly limited to read-only or write-only access.

Shape Specification

A *shape* specification is used when the base is not scalar. It describes array indexing or a structure.

A structure shape is specified with a *structure_id* enclosed in angle brackets.

An array specification implies the procedure or parameter is an array. It follows the same rules as a definition of an array.

The Procedure Body

The statements in the procedure (the *body*) define the actions and internal variables used. In place of these statements, the reserved word **pragma** can be used. It has parameters which indicate the implementation is defined by another language or by externally provided code, or code known to the compiler, or it may indicate conditions governing the compilation, such as inline code generation.

Generator Functions

A generator function is defined with the keyword **repeat** as a *proc_option*.

It can be used only in a **for** construct or the right side of an array assignment.

A generator function returns ("yields") a value using the **return** statement. Each time it yields a value, the current state is preserved, and when invoked again, it resumes where it left off. It finally quits, indicating an end to the set of values, when it encounters **return nil** or the function end, which then returns an indication that it is done.

A generator is a function, not a method.

A generator can be used as an array of the values returned.

Range

A generator function **Range** is supplied. It returns a sequence of **int.64** values. It is used with a base specified, with these two optional **int.64** parameters:

start	default is 0.
by	default is 1; nonzero; if negative, the base (stopping value) must be <= **start**.

The base is required, it is the stopping value. Range stops if at or beyond this value in the direction implied by the sign of **by**.

If these are invalid, Range is unable to produce a value, and reports an invalid range.

Range can be used as an array of the values returned.

Examples:

```
int.64[] ABC
ABC = 10.Range                  # sets 10 array elements to 0 through 9
ABC = 11.Range(start=1)         # sets values 1 through 10
ABC = 12.Range(start=2, by=3)   # values 2,5,8,11 but not 14
ABC = (-6).Range(by=-1,start=5) # values 5,4,...,-4,-5
```

A combination of Range and an *inline for expression* can be used for "list comprehension", as in this example:

```
int[] sels
sels[] = (for x from 101.Range if x^2 > 3)*2 # yields 4, 6, 8, 10 ... 200
```

Each

The Each function is a generator. It can be used on a string or array or any traversable object.

Invocation of a Procedure

A procedure defined on a given type is invoked by naming an object of that type, then the period (selector character), then the procedure name with parameters as needed.

Invocation of a method will not use parentheses around the parameters.`

A function invocation does not use a pair of parentheses when it has no parameters.

If there is no base_type the base and the period are omitted.

When a function returns a reference to an object, that object may have other procedures or variables. A function invocation can then be followed by a selector period and another reference. Invocations can be nested/stacked.

The *base* object (on xyz.MethodCall for example, the base is xyz) acts as if it is passed as a parameter named **$**, which is passed by reference. When the base is a constant, a basic type value or an expression in parentheses, a reference to a copy is passed.

When an expression of any kind is passed as a parameter, a copy is passed rather than a reference to the original value. This applies even for a parenthesized item, or for type casts, or for a unary plus. In other words, **(abc)** is a copy of **abc**, as is **+abc**.

A structure or defined-type object or array or matrix is passed as a parameter, a reference is passed.

When an array or matrix element (member) is passed, the value is passed, not a reference.

A string or any basic type, passed as a parameter, is passed as a value or copy, not a reference.

Built-in Object Procedures

These procedures can be applied to any object, variable, stack or array, treating them as objects.

Table: Built-in Object Procedures

Procedure	Kind	Result
`Clear`	method	the named object is destroyed
`Copy`	function	a copy of the item
`getIndexType`	function	string, the type name of an associative array's index
`getName`	function	string, valset constant name for value in variable
`getType`	function	string, name of the type of the object
`getValue`	function	**uint** value of the valset constant
`getValuesArray`	function	returns string-indexed array of values in the valset type used as base
`isNil`	function	**true** if the base reference is **nil**
`isNotNil`	function	**true** if the base reference is not **nil**
`isType(string s)`	function	**true** if the object type or its inherited type matches **s**

Type names are returned in the same case they are defined. In the case of a valset, it returns "valset typename", a defined object as "object typename".

The functions **getName**, **getType**, **getIndexType**, **getValue** and **getValuesArray** are reflection functions or get properties. Reflection in CB is the ability to retrieve attributes of an object.

Name References in an Object

Names within an object or inherited in an object are referenced without need to imply they refer to the current instantiation or static item. If a name overrides the same name, it is possible to refer to the parent's version by prefixing it with a selector period and the object name inherited.

Example:

```
{ object ABC
  string x = 'Hello'
}

{ object DEF uses ABC
  string x = 'World'
  { new # constructor
    @ ABC.x | ', ' | x | '!'
  }
}

# use it -
DEF hi  # prints: "Hello, World!" on instantiation
```

Within an object, a reference to the whole object is written as the keyword **$**.

Prototypes

Prototypes define the usage of procedures, operators and type casts. An inherited prototype must be defined as a procedure at a subsequent level of object or structure. They are used in in abstract object and structure definitions, and are permitted in non-abstract definitions, indicating a requirement. The *base_type* is not shown in a prototype.

A prototype begins with a question mark. A property cannot be a prototype.

Function Prototype

? [*proc_option*] **func** *name* [**(***param_spec , ...***)**] **gives** *type*

Method Prototype

? [*proc_option***] method** *name* [*param_spec , ...*]

Type Cast Prototype

? func gives *type*
# note that there is no name

Operator Prototype

? func *operator* [*param_spec*] **gives** *type*
# note there are no parentheses

Pragma Declaration

The **pragma** reserved keyword introduces a declaration which informs the compiler about how to compile code. Following the keyword **pragma** are special options and code generation instructions. Some of these may be restricted only to source from standard directories or not available except in standard code.

Some features to be implemented:

```
pragma use # bring in precompiled objects – NOT source code, and
       not allowed inside objects, procedures, or any definition
pragma adapt [function] – using as a procedure body allows any code
pragma inline # makes this procedure expand inline, no procedure call used
```

Certain behavior defaults for generated code or optimizations are controlled by pragma:

```
pragma option ...
pragma unroll
```

Printing Strings

The standard library **std** defines some built-in procedures. One method which is defined is named **@**. This method takes a single parameter, a string, and writes it to the standard output stream or back to the connected browser or other connection with end-of-line character(s) added to the end of the output string.

Example:

```
string helper = 'Watson'
@ "Come help me, {helper}, I need you!"
```

Note the string insertion in the message.

These method names are not reserved words. They are defined as single string parameter methods with no base type specified.

Hello, World! Program

The standard "Hello, World!" program in CB is very simple:

```
{ object @Main
   { new
     @ "Hello, World!"
   }
}
```

It is automatically invoked by its constructor:

```
@Main
```

The constructor can have parameters.

Example

The example shows an object type named Animal, inherited by objects named Cat and Dog:

```
# the object Animal - - -
{ object Animal
  string animal_name = 'unnamed', sez = 'I am an animal.'
  { method Speak
    @ "{$.getType} {animal_name} '{sez}'"
  }
}
# animals Cat and Dog
{ object Cat uses Animal # overriding method Speak for Cat - - -
  { method Speak
    @ "{$.getType} {animal_name.Caps} says \"Meow!\""
  }
}
{ object Dog uses Animal # overriding Speak for Dog
  { method Speak
    @ "{$.getType}: {animal_name.Upper} says \"I am a canine.\""
  }
}
# a snake -
{ object snake uses Animal
}

# the program - - -
{ object @Main
    { new
      Cat Fluffy = Cat(animal_name='FLUFFY'), Tom = Cat(animal_name='tom cat')
      Dog Fido = Dog(animal_name='Fido')
      snake Coral(sez='s-s-s-s')
      Animal Buddy
      Animal pets[] = Animal[Fluffy, Tom, Fido, Buddy, Coral] # an array
      { for who from pets
        who.Speak
      }
    }
}
# expected output -
# Cat Fluffy says "Meow!",
# Cat Tom Cat says "Meow!"
# Dog: FIDO says "I am a canine."
# Animal: unnamed 'I am an animal.'
# snake: "s-s-s-s"
```