Struct Proc 9a Tutorial Guide

Joseph Phillips

Applied Philosophy of Science

Last Modified 2024 July 8

I. Introduction

Congratulations! You are about to learn the basics of the Struct Proc 9A computer language, the low-level internal language of the SP9a reasoner. Learning this language will let you talk to the knowledge base directly, and create your own reasoning methods.

From the culture of scientific mathematics!

A quick note: the design of this language leans heavily on both the notation and culture of basic scientific mathematics. The language has features not common to other languages, even scientific computational languages. When language design decisions were made, they were made with the notation and culture of basic scientific mathematics in mind. This document will be sure to highlight these decisions with asides like this one.

Also note, this document is only a tutorial. Its purpose is to give you an intuition for the design of the language, and to get you to start writing useful code. However, there are many details that it does not explain. For a more thorough and systematic discussion of the language, please see "The SP9A Reference Manual".

A word about font usage.
Emphasis is denoted by *underlining and italicizing*.
Text that could be typed by the user is denoted in *`italicized courier font`*.
Text that could be returned by StructProc9A is denoted in `non-italicized courier font`.
Discouraged StructProc9A code is in **`orange courier font`**.
Illegal StructProc9A code is in **`red courier font`**.
Know bugs are shown in **`blue courier font`**.

II. How to start Struct Proc, and how to leave

Start StructProc by simply typing `sp9a` from the Linux command line:

```
$ sp9a
```

or perhaps,

```
$ sp9a path=(path of StructProc9 home directory)
```

The program will respond by telling you the language file it will use, and then listing the library files it loads. Finally it will give you the command prompt:

```
Reading kb file defaultInitKb.som9
SP9a1 :) [1]
```

Now that you have got it working, among the most important commands of any system is the request to leave. To leave the StructProc9 type `quit`, followed by a semicolon (`;`).

```
SP9a1 :) [1] quit;
quit
$
```

The semicolon is important: all commands need to end with one. If you forget the semicolon then type it on the next line.

```
SP9a1 :) [1] quit
SP9a1 :) [2] ;
quit
$
```

The Scienceomatic is a whole environment having several interacting programs. Besides running a single process with a command-line interface, perhaps the other most popular option is to run it with an HTTP interface. To do that, type on the Unix command line:

```
$ ./somAdmin --serverMode=http
http (2022-01-10 11:08:00): Beginning
Reading kb file defaultInitKb.som9
http (2022-01-10 11:08:00): Connect to http://127.0.0.1:8080
som (2022-01-10 11:08:00): Beginning
som (2022-01-10 11:08:00): Process 2242 attempt
execl(/opt/AppPhiloSci/SOM/Ver9A/Bin/somRApp)
Running on port 58384.  Press Ctrl-C to stop:
```

Leave the command-line shell running for as long as you want the server to keep running. Connect to this process by loading the URL `http://127.0.0.1:8083` into your browser. Like the introductory message says, press `Ctrl-C` in command-line window to stop the Scienceomatic environment.

Only one StructProc9 application is allowed to run in a given directory at a given time. This is enforced in part by the creation of a file named `lock.txt` in the `/Misc` directory of the process. If the process crashes then be sure to delete this file before attempting to run it again.

III. Basic Types and Operations

Like most computer languages, Struct Proc 9A has integers and floating points (which, following Pascal, are called “reals”, or `Real`) predefined. The common mathematical operations are defined on them, too. You can see this from the command line:

```
SP9a1 :) [1] 1+2;       // Addition of 2 integers
3
SP9a1 :) [2] 1.+2.;     // Addition of 2 reals
3.
SP9a1 :) [3] 1+2.;      // Addition of real and integer give real
3.
SP9a1 :) [4] 4.5 * 2;
9.
SP9a1 :) [5] 4 - 7;
-3
SP9a1 :) [6] sin(3.14159265358979323846);
1.22465e-16             // sin() expects a real in radians
                        // The result should be 0.0, but this is
                        // close.
```

But wait! Beyond having mere classes `Integer` and `Real`, the language has classes `Rational` and `Complex` too:

```
SP9a1 :) [7] 4/7;       // 4 divided by 7 is the fraction 4/7
4\7                     // This is how to represent that rational
SP9a1 :) [8] (4/7)*7;   // Multiply it by 7 and get the integer 4
4
SP9a1 :) [9] 40/70;     // Common primes are automatically removed
4\7                     // from numerator and denominator
SP9a1 :) [10] exp(3.14159265358979323846i);
(-1.+1.22465e-16i)      // Well, exp(pi*i) should be -1.,
                        // but +1.22465e-16 is close to 0.0
```

There are also `true` and `false` of class `Boolean`:

```
SP9a1 :) [11] true and false;
false
SP9a1 :) [12] true or false;
true
SP9a1 :) [13] !false;
true
```

And, of course, there is class `String`.

```
SP9a1 :) [14] "Hello " ++con "there";  // ++con is the concatenation
Hello there                            // operator
```

Numeric operators include:

Operator	Associativity:
(unary operators)	N/A
^	right-to-left
*** /**	left-to-right
+ -	left-to-right
< <= > >=	left-to-right
==ref !=ref ==num !=num	non-associative

When used as a *binary* operator, `^` means raise-to-the-power. The `*` and `/` naturally mean multiplication and division. The `+` and `-` naturally mean addition and subtraction. The operators `<`, `<=`, `>` and `>=` mean their respective comparisons, and all return `Boolean` values. The `Boolean`-returning operators `==ref` and `!=ref` compare if the two operands refer to the same thing. The `Boolean`-returning operators `==num` and `!=num` compare if the two operands are numerically equal. Thus:

```
SP9a1 :) [15] 2 ==num 2.0;
true
SP9a1 :) [16] 2 ==ref 2.0;
false
SP9a1 :) [17] 3\2 !=num 1.5;
false
SP9a1 :) [18] 3\2 !=ref 1.5;
true
```

Some predefined functions include:

Name:	Purpose:
`sin(x)`	Sine of `x` radians
`cos(x)`	Cosine of `x` radians
`tan(x)`	Tangent of `x` radians
`asin(x)`	Arc-sine of `x` returning radians
`acos(x)`	Arc-cosine of `x` returning radians
`atan(x)`	Arc-tangent of `x` returning radians
`exp(x)`	e^x
`exp2(x)`	2^x
`exp10(x)`	10^x
`log(x)`	Logarithm base e of `x`
`log2(x)`	Logarithm base 2 of `x`
`log10(x)`	Logarithm base 10 of `x`

IV. Annotated Values

So far the numbers we have seen, instances of `Integer`, `Rational`, `Real` and `Complex`, have been *non-annotated*. That means they have not been given meta-data to describe anything; they are just numbers.

Annotated values have meta-data to describe what the numbers mean, and how they can properly be used. Annotated values are instances of `AnnotatedValue`. An example of an `AnnotatedValue` instance is the one below for equatorial radius of the Earth:

```
6378.1366 (* +/-0.0001, defaultKmDomain *);[1]
```

Annotated values are denoted by ordinary rational or floating point numbers followed by annotations delimited by `(*` and `*)`. Here, the floating point number `6378.1366` has been annotated with two the attributes:

- `+/-0.0001:` This is the uncertainty associated with the value. The system takes this as the standard deviation.
- `defaultKmDomain`. As hinted by its name, the domain tells the units (kilometers). Therefore, it also implies the dimensionality of the value: `Distance`. Domains may also tell which values are legal for an `AnnotatedValue` instance.

All annotated values have domains associated with them. StructProc9, however, allows users to be less precise and give units instead. When this is done, StructProc9 looks up the default domain of the units (attribute `unitsDefaultDomainA`), or creates the corresponding `Domain` instance if necessary. Thus, these are legal:

```
3.14159(*meters*);
3.14159(*kilometers*);
3.14159(*millimeters*);
```

Further, StructProc9 also has a parser just for units, and the nicknames of common units have been specified. Units may be given as a string of operations on abbreviated units names. Thus, these are all legal as well:

```
3.14159(*"m/sec"*);
3.14159(*"m/sec^2"*);
3.14159(*"1/sec"*);
3.14159(*"m*kg/sec"*);
```

Strings may specify units according to the following rules:

(1) Units may be given by their official names (*e.g.* `"meters"`), or by their specified abbreviations (*e.g.* `"m"`). Only units of fundamental dimensions may be specified. Thus, even though the system knows of Newtons and its abbreviation `"N"`, this is illegal: `"N*meters"`. Instead say `"kg*m^2/s^2"`, or `"Joules"`, or just `"J"`.

(2) Two or more units may be multiplicatively combined with the asterisk, as in `"meters*m"`.

---

1 From “IAU Division I Working Group, Numerical Standards for Fundamental Astronomy , Astronomical Constants : Current Best Estimates (CBEs)” http://maia.usno.navy.mil/NSFA/NSFA_cbe.html#EarthRadius2009, downloaded 2019 February 3.

(3) Powers may be specified by the unit, followed by the caret, followed by a non-zero integer. An example is `"m^2"`.

(4) Inverse relationships may either be specified by following a caret with a negative integer, or by giving the units after a forward slash. So both `"m/sec"` and `"m*sec^-1"` are legal. Negative powers may not be given on the denominator side, so `"1/m^-1"` is illegal.

(5) If there is no numerator unit, give `"1"` as the numerator, as in `"1/sec"`.

(6) The forward-slash, if it appears at all, may only appear once.

(7) Putting units both as numerator and denominator is allowed, but discouraged. Thus, these are discouraged: `"meters/m"`, `"m*m^-1"`.

(8) Raising units to the power of zero is also allowed but discouraged. So this `"m^0"` is discouraged.

Please note, however, that while automatically-created domains have dimensions and units, they lack meaningful limitation information. Thus, specifying the domain by name is always preferred.

The system is clever enough to convert units when adding or subtracting:

```
BUG BELOW!  Wrong number of returned digits are returned:
SP9a1 :) [6] 1.000(*meters*) + 1.0(*cm*);
1.0e+00(*defaultMetersDomain*)
SP9a1 :) [7] 1.0(*cm*) + 1.000(*meters*);
1.0e+02(*defaultCmDomain*)
```

From the culture of scientific mathematics!

Here, the resulting sum (or difference for subtraction) will get the domain (and thus units) of the first number of the addition (or subtraction). This follows from the culture of scientific mathematics:

$$x[new] = x[old] + \Delta x$$

`x[new]` is just a slight perturbation from `x[old]`. So, `x[new]` inherits the meta-data of `x[old]`, not that of Δx.

From the culture of scientific mathematics!

Also note, that this system keeps track of the number of significant digits. Therefore, the sum:

```
1.0000(*meters*) + 1.0(*meters*)
```

returns:

```
2.0e+00(*defaultMetersDomain*)
```

as opposed to:

```
2.0000e+00(*defaultMetersDomain*)
```

or some other value.

A note about notation: the system has to distinguish between attributes and the dimensions of their intended values. For example, we believe the Earth has mass of 5.972168×10^{24} kg (or, `5.972168e+24(*kg*)` in our notation). The term “mass” in “Earth has mass” serves as an attribute of Earth. However, the value `5.972168e+24(*kg*)` has dimensionality of Mass. One is an attribute of a physical thing, the other is an attribute of a number.

By convention, we denote attributes by beginning them with a lowercase letter. Thus, the attribute of Earth is `mass`. We denote the corresponding dimension with by beginning them with an uppercase letter. So, the dimensionality of `5.972168e+24(*kg*)` is `Mass`.

V. Instances in the knowledge base, and running methods

The language is a true object-oriented language where everything is either an instance or a class. Also, dynamically-generated classes become full fledged classes with all the properties of other classes.

Use the C++ arrow notation, `instance->method()`, to run a method on a class. For example, to ask to what class anything belongs, run the method `localGet()` with argument `instanceOf`. To ask what is the superclass of any class, run method `localGet(isA)`.

```
SP9a1 :) [14] 1->localGet(instanceOf);
Integer
SP9a1 :) [15] Integer->localGet(isA);
Rational
SP9a1 :) [16] 1->localGet(instanceOf)->localGet(isA);
Rational
SP9a1 :) [17] 1.->localGet(instanceOf);
Real
SP9a1 :) [18] 4\7->localGet(instanceOf);
Rational
SP9a1 :) [19] i->localGet(instanceOf);
Complex
SP9a1 :) [20] "Hello world"->localGet(instanceOf);
String
```

There are several other predefined methods for different classes. For example, instances of class `String` have:

```
SP9a1 :) [21] "hello"->string_length();
5
SP9a1 :) [3] "hello"->string_capitalize();
Hello
SP9a1 :) [4] "hello"->string_hasPrefix("he");
true
SP9a1 :) [5] "hello"->string_hasPrefix("she");
false
SP9a1 :) [6] "hello"->string_prefix(3);
hel
```

Here are some more methods on basic types.

String method:	Purpose:
`rational_numerator()`	Returns the numerator integer of a rational number.
`rational_denominator()`	Returns the denominator integer of a rational number.
`string_length()`	Returns the length of the string.

String method:	Purpose:
`string_capitalize()`	Returns the same string with its first character capitalized. If the string is empty, or if its first letter is not a lowercased letter, then returns the same string. Leaves all other characters but the first unchanged.
`string_hasPrefix(substring)`	Returns true if the given string begins with `substring`, or false otherwise.
`string_prefix(count)`	Returns the first `count` characters of the string.
`string_hasSuffix(substring)`	Returns true if the given string ends with `substring`, or false otherwise.
`string_suffix(count)`	Returns the last `count` characters of the string.
`string_substring(index,count)`	Returns the substring of the `count` characters starting from index `index`.

Now is a good time to introduce to pre-opened output files, `stdOut` and `stdErr`. The main methods on them are `print()` and `printLn()`, both of which take a single argument:

```
SP9a1 :) [7] stdOut->print("Hello");
HellostdOut
SP9a1 :) [8] stdOut->printLn("Hello");
Hello
stdOut
```

Notice that both return the same file that they were given. The reason for this is to allow for expresions like:

```
SP9a1 :) [2] stdOut->print("Hello ")->print("there")->printLn("!");
Hello there!
stdOut
```

VI. Lists and Bags

The language supports lists using common mathematical notation:

```
SP9a1 :) [1] [0, 1, 2, 3, 4];
[0,1,2,3,4]
```

From the culture of scientific mathematics! This is the same as lists are represented in mathematics.

These lists are implementation as vector lists, as you may easily see:

```
SP9a1 :) [2] [0,1,2]->localGet(instanceOf);
VectorList
```

Node lists may be made by enclosing the items between `[**` and `**]`.

```
SP9a1 :) [3] [** 0,1,2 **]->localGet(instanceOf);
NodeList
```

Of course, the difference between the two is the classic ease-of-random-access (advantage: `VectorList`) vs. ease-of-insertion/deletion-in-the-middle (advantage: `NodeList`). Insertion and iteration are covered below.

It supports bags, too:

```
SP9a1 :) [10] {0, 1, 2, 3, 4};
{0,1,2,3,4}
```

From the culture of scientific mathematics! This is taken from set notation in mathematics.

A bag is like a set in that there is no implied order among included things. However, a bag is like a list in that the same thing may be included more than once.

```
SP9a1 :) [11] {0, 1, 2, 3, 4, 0, 1, 2, 3 };
{0(2),1(2),2(2),3(2),4}
```

Here, the index notation after each included item tells how many times that that item is included (if the count is 2 or greater).

To force bag to act like a set, where all items are included at most once, write `[*true*]` immediately after the set:

```
SP9a1 :) [7] {0,1,2,2}[*true*];
{0,1,2}[*true*]
```

We will see why we use this notation in the next chapter.

The last basic data-structure is that map. Each mapping has form `key(value)`, and they are enclosed between `<<<*` and `*>>>`.

```
SP9a1 :) [8] <<<* methyl(1), ethyl(2), propyl(3) *>>>;
<<<*methyl(1),ethyl(2),propyl(3)*>>>
```

To get a mapping, use the `map_get()` method:

```
SP9a1 :) [1] <<<*methyl(1),ethyl(2),propyl(3)*>>>->map_get(propyl);
3
```

To insert or change a mapping, use the `map_put()` method:

```
SP9a1 :) [1] <<<*methyl(1),ethyl(2),propyl(3)*>>>->map_put(butyl,4);
<<<*methyl(1),ethyl(2),propyl(3),butyl(4)*>>>
```

The (basic) methods of data-structures are:

Method	Description
`dataStruct_isEmpty()`	For all: returns `true` if the data structure is empty, or `false` otherwise.
`dataStruct_size()`	For lists: returns the number of items in the data structure. For bags: returns sum of the number of times all inserted identities are present. For maps: returns the number of keys in the map; each <key,value> pair only counts once.
`dataStruct_distinctCount()`	For lists: returns number of distinct identities. For bags: returns number of distinct identities, not summing their counts. For maps: same as `dataStruct_size()`.
`dataStruct_iter()`	For lists: returns an iterator to range over the identities in the list. For bags: returns an iterator to range over the identities in the list; if an identity `c` is in the bag `N` times then the iterator will return `c` `N` times consecutively. For maps: returns an iterator to range of the pairs. Iterator method `iter_key()` returns the key of the pair. Iterator method `iter_value()` returns the value.
`dataStruct_insertA(item)`	For lists: inserts `item` at the beginning of the list. For bags: inserts `item` (again, if already present). For maps: throws exception (maps want both a key and value) For lists and bags: returns the data structure.
`dataStruct_insertZ(item)`	For lists: inserts `item` at the end of the list.

Method	Description
	For bags: inserts `item` (again, if already present). For maps: throws exception (maps want a key and value pair) For lists and bags: returns the data structure.
`dataStruct_insert(item)`	For lists: inserts `item` at the end of the list. For bags: inserts `item` (again, if already present). For maps: throws exception (maps want both a key and value) For lists and bags: returns the data structure.
`dataStruct_insert(item,n)`	For bags: inserts `item` `n` more times. For lists and maps: throws exception.
`dataStruct_didInsertBecauseNotPresent(item)`	For lists: inserts `item` at the end only if it is not already present. For bags: inserts `item` only if it is not already present. For maps: throws exception. For lists and bags: returns the data structure.
`dataStruct_doesHave(item)`	For lists and bags: returns `true` if the data structure has `item` at least once, or `false` otherwise. For maps: returns `true` if the data structure has `item` as a key, or `false` otherwise.
`dataStruct_didRemove(item)`	For lists: Returns `true` if the first occurrence of `item` was found and removed, or `false` if there are none. For bags: Returns `true` if the one occurrence of `item` was removed, or `false` if there are none. For maps: Returns `true` if the pair with key `item` was removed, or `false` if there was none.
`dataStruct_clear()`	For all: removes all items in the data structure, making its size 0. Returns the data structure.
`dataStruct_copy()`	For all: returns a copy of the data structure.
`list_firstItem()`	For lists: returns first identity, or `null` if the list is empty. For bags and maps: throws exception.
`list_secondItem()`	For lists: returns second identity, or `null` if the list is less than two items long. For bags and maps: throws exception.
`vList_didPut(index,item)`	For vector lists: places `item` at integer indexed position `index` if it is a valid index into the vector list, and returns `true`. If `index` is not a valid index into the vector list then returns `false` and takes no further action.

Method	Description
`vList_get(index)`	For vector lists: returns the item at integer indexed position `index` if it is a valid index into the vector list. Returns `null` if `index` is not valid.
`vList_numInsertA()`	For vector lists: returns the number of times `dataStruct_insertA(item)` has been called since the last time `dataStruct_clear()` has been called.
`bag_count(item)`	For bags: returns the number of times `item` is present.
`map_put(key,value)`	For maps: inserts <`key`,`value`> pair, overwriting the existing value for `key` if it already is matched with a value. Returns map. For lists and bags: throws exception.
`map_get(key)`	For maps: returns the value to which `key` has been paired, or null if there is no such value. For bags and lists: throws exception.
`list_sort(order)`	For vector lists and node lists: sorts the list (must be of either `Number` or `String` instances), according to order (which must be either `ascendingOrder` or `descendingOrder`).
`list_sort(order,attribute)`	

VII. Explicit construction

So far we have seen

Data-Structure:	**Example:**
`VectorList`	`[ 0, 1, 2, 3 ]`
`NodeList`	`[** 0, 1, 2, 3 **]`
`Bag` (and restricted to set)	{ 1,
`Map`	`<<<*0("zero"),1("one"),2("two"),3("three")*>>>`

However, we have not considered *structure* construction, the *Struct* in StrucProc. We do so now.

The knowledge base keeps track of the knowledge associated with concepts. Distinct concepts have are represented by their own *identities* internally. With each identity are <attribute, value> pairs, or *properties*.

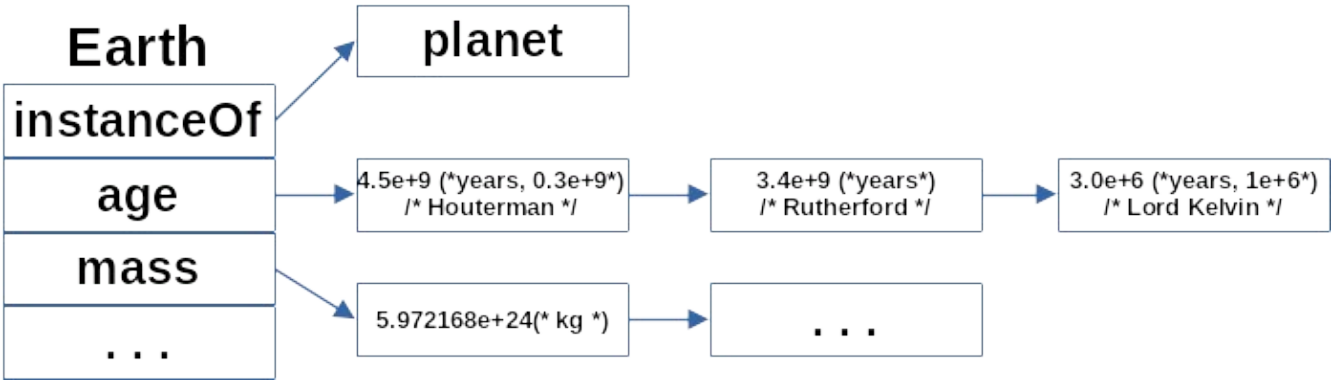
StructProc9 was designed for science. Often in science we want to remember several values for attribute. Sometimes this is because we may now have a most believed modern value, but science assumed different values in the past. Other times this is because different values were obtained by different methodologies.

For example, Western Science has had a number of different estimates for the age of the Earth.

- 6000 years (Ussher)
- 75 Kya (Buffon)
- “several billion” (de Maillet, Buffon)
- ∞? (Hutton, Lyell)
- 100 Mya (Lord Kelvin)
- 20-40 Mya (Lord Kelvin)
- 3.4 Gya (Rutherford)
- 4.6 Gya (Meyer)
- 4.5±0.3 Gya (Houterman)

Keeping all of these around lets us re-do science within the worldview of historical science contexts.

Although some attributes must be single value, most may have a list of values:



StructProc9 allows for the managing of and iteration over these lists of values.

Multiple (perhaps conflicting) values of for the same attribute may be introduced all at once using *explicit construction*.

```
Earth
{*
  instanceOf->assertZ(planet),
  mass->assertZ(5.97237e+24(*defaultKilogramsDomain*)),
  radius->assertZ(6378.137(*km*)),
  circumference->assertZ(40075.017(*km*)),
  age->assertZ(4.5e+9(*years,0.3e+9*)),// Houterman
  age->assertZ(3.4e+9(*years*)),       // Rutherford
  age->assertZ(3.0e+6(*years,1e+6*))   // Lord Kelvin
*};
```

Here, `instanceOf`, `mass`, `radius` and `age` are attributes. All of them have values ***asserted***. Asserting knowledge using explicit construction uses the same arrow (`->`) syntax used to call methods.

From the culture of scientific mathematics!

Notice the comma-separated assertions within the set-like {* and *} delimiters! This is purposeful.

In mathematics the sets `{1,2,3}` and `{3,2,1}` are equivalent. Likewise, asserting the properties of ***different*** attributes in any order constructs the same description.

Although asserting the properties of ***different*** attributes in any order constructs the same description, asserting `instanceOf` first is preferred for ready comprehension.

```
//  Preferred: instanceOf first
Earth
{*
  instanceOf->assertZ(planet),
  age->assertZ(4.5e+9(*years*))
*};
```

```
//  Discouraged: instanceOf
//  not given first
Earth
{*
  age->assertZ(4.5e+9(*years*)),
  instanceOf->assertZ(planet)
*};
```

Assertion ordering does matter for asserting different properties of the ***same*** attribute. For instances (as opposed to classes), the three main assertion methods come from Prolog and are:

- `attribute->assertA(value)`: Place `value` at the beginning of the value list of `attribute`.
- `attribute->assertZ(value)`: Place `value` at the end of the value list of `attribute`.
- `attribute->assert(value)`: Erase the previous list for `attribute` and make `value` is sole value.

Attribute `instanceOf` is special because it places an identity in the knowledge base. Multiple inheritance is supported, but discouraged:

```
Earth
 {*
   instanceOf->assertZ(planet),              // Multiple inheritance
   instanceOf->assertZ(lifeSupportingBody), // is discouraged
   ...
 *};
```

Within the same structure `assertZ()` is preferred over the same explicit construction because the user sees the assumed best value first:

```
//  Preferred                          //  Discouraged
Earth                                  Earth
{*                                     {*
  age->assertZ(4.5e+9(*years*)),         age->assertA(3.0e+6(*years*))
  age->assertZ(3.4e+9(*years*)),         age->assertA(3.4e+9(*years*)),
  age->assertZ(3.0e+6(*years*))          age->assertA(4.5e+9(*years*)),
*};                                    *};
```

One exception to the above rule is when on subsequent explicit constructions elaborate and refine knowledge about an identity over time:

```
// Encouraged
Earth
{*
  age->assertZ(3.0e+6(*years*))  // Lord Kelvin
*};
...

Earth
{*
  age->assertA(3.4e+9(*years*))  // Rutherford
*};
...

Earth
{*
  age->assertA(4.5e+9(*years*))  // Houterman
*};
```

VIII. Implicit Construction

Because explicit construction suffers for verbosity there are also several forms of *implicit construction*. Here the attribute does not have to be given because it is implicit in the ordering of the expression.

We have already seen an example of implicit construction when we defined a set (as opposed to a bag):

```
SP9a1 :) [12] {1,2,3,4,5,3,2,1}[*true*];
{1,2,3,4,5}[*true*]
```

The argument `true` is passed to the `Bag` constructor because it is between the `[*` and `*]`. The first argument (if given) to the `Bag` constructor is used to set the bag attribute `isBagRestrictedToSet`. The default value that the constructor sets for this attribute is `false`, but the user may override this by giving `true` in the implicit constructor call.

From the culture of scientific mathematics!

Notice the similarity between list notation in mathematics (`[ .. ]`) and the implicit construction delimiters of `[* … *]`. This is because ordering matters in both cases.

Implicitly constructing an instance of a class requires that that class has a *constructor* that is inherited by its instances.

There are several variations on implicit notation. One can create a named instance of a class. For example, in addition to classes StructProc9, also has `EnumerativeCluster`. Instances of `EnumerativeCluster` exist to group like things together, but without the ontological certainty implied by making them all members of the same class. `EnumerativeCluster` is itself a class its constructor expects a class as its first argument (every member but be an instance of the class) and a `VectorList` as its second argument (and here are the current members).

Thus we can define a terrestrial planet as "*a planet, of which Earth is an example*":

```
`terrestrial planet` [* EnumerativeCluster | planet, [Earth] *];
```

Here, the backticks in `` `terrestrial planet` `` only serve to group characters of a name that also includes a space.

From the culture of scientific mathematics!

Notice the syntax:

```
name [* class | constructor arguments *]
```

Here, the vertical bar is meant to connote "such that" from mathematics. The whole thing is meant to be read as "*`name` is an instance of `class` such that it has properties given in the `constructor arguments`*".

So, the example above, we have "*`` `terrestrial planet` `` is an instance of `EnumerativeCluster` such that its members must be instances of class `planet`, and the one member we have so far is `Earth`*".

Another variation of implicit construction is the creation of *anonymous instances*: identities with properties, but without names. Its syntax is similar to used by named implicit constructions, but it moves the class name before the `[* … *]`, and precedes everything with `^`:

```
^class [* constructor arguments *]
```

Not really from the culture of scientific mathematics!

The caret is an editing mark meaning "insert punctionation/word/phrase here". In a sense, that is what is it doing here: inserting a new compile-time instance of a class.

So, if one wanted an anonymous `EnumerativeCluster` that included `Venus` and `Mars`, one would say:

```
^EnumerativeCluster[* planet, [Venus, Mars] *];
```

Very important! Using the caret makes a new item at *compile-time*. This means one and only one item will be created when the code compiles. Thus, it will live for the life of the knowledge base.

To dynamically create new items as the code runs use `new`. The `new` operator will be introduced in the next chapter.

IX. Flow control, Variables and the new operator

A) Flow control

This chapter assumes that you have seen *anonymous implicit construction*. If not, please read the previous chapter.

Now we can write code! Recall, conceptually *everything* in StructProc9 is an item in the knowledge base. This includes snippets of code. Code uses anonymous implicit construction syntax. Conceptually, all flow control elements are instances of classes.

If statements take either 2 or 3 arguments, depending upon whether there is code for "else":

```
^If[* condition, then *]
^If[* condition, then, else *]
```

In StructProc9, all code returns the last thing that was computed. This includes If-statements. Thus, the best translation of `^If[* condition, then, else *]` into C is:

```
( condition ? then : else )
```

There are 3 types of loops: while-loops execute the "test" expression before the "body", repeat-loops execute the "body" before the "test" (akin to `do-while` loops in C, and `repeat-until` loops in Pascal), and for-loops do the "initialization", "test", "body", and "increment" in that order.

```
^While[* test, body *]
^Repeat[* body, test *]
^For[* initialization, test, body, increment *]
```

In C when we want to omit an element of a for-loop, we just omit it, as in:

```
for ( ; i <= 10; ) {
  printf("%d\n",i);
  i++;
}
```

However, in StructProc9 we have to give something. Except for "test", it is the culture of StructProc9 to give `null` for arguments that ought to be ignored. So the equivalent StructProc9 code is:

```
^For
[*
  null,
  @i <= 10,
  ^Do[*[ stdOut->printLn(@i), @i := @i + 1 ]*]
  null
*]
```

We should also introduce `Do`, which defines a block of code to execute serially. It serves as `{ … }` in C, and `begin … end` in Pascal.

```
^Do[* codeList *];
```

Because `Do` wants only one thing (a list) and because the list always is denoted with […], it is the culture of StructProc9 to group the syntax together as `[*[ … ]*]`:

```
//  Recommended syntax:
^Do
[*[
  stdOut->printLn(@i),
  @i := @i + 1
]*]
```

Notice that all are preceded by the `^`. This is because (conceptually) `If`, `While`, `Repeat`, `For` and `Do` are all classes. All we are doing is making anonymous instances of them.

B) Variables

Now that have seen `Do`, we can define variables and constants. For the most part, both must be declared within the scope a `Do`. For variables there are 3 forms:

```
^VarDecl[* @var *]
^VarDecl[* @var, Class *]
^VarDecl[* @var, Class, expression *]
```

Starting with the ***last*** one, it declares variable `@var` to be limited to class `Class`, and to have the initial value computed by `expression`.

The middle one specifies the class of `@var`, but not the initial value. It will be given the default value for the class:

- `false` for `Boolean`,
- `0` for `Integer` and `Rational`,
- `0.0` for `Real` and `Complex`,
- the empty string for `String`,
- `null` for anything else.

The first one does the same as the second, but the class defaults to `Idea`.

Declaring constants is similar, but all three arguments must be specified:

```
^ConstDecl[* @pi, Real, acos(-1) *]
```

We have variables! Let's use them. We can count to ten:

```
^Do
[*[
  ^VarDecl[* @i, Integer, 1 *],
```

```
  ^For
  [*
    null,
    @i <= 10,
    stdOut->printLn(@i),
    @i := @i + 1
  *]
]*];

1
2
3
4
5
6
7
8
9
10
stdOut
```

Notice several points:

- `@i` was initialized in `^VarDecl`, so the initialization step of `^For` is `null`.
- The code is wrapped between [*[and]*]. Recall, this is really [...] inside of [*...*]. However, it is best to think of it as combined syntactical elements.
- The code counts from `1` to `10`, as expected. Then it prints `stdOut`. This is because `Do` returns the last thing that it computed, which was the last thing that `For` computed, which was the last thing that the body computed, which was what `printLn()` returned, which was the file to which it printed.
- Everything in the constructor lists for `VarDecl`, `For` and `Do` are all separated by commas, because they are in lists. However, at there is a semicolon at the end of everything.

And now we can also iterate over data-structures:

```
^Do
[*[
  ^VarDecl
  [*
    @iter,
    Iterator,
    ["StructProc","is","a","strange","language"]->dataStruct_iter()
  *],

  ^For
  [*
    null,
   !@iter->iter_isAtEnd(),
    stdOut->printLn(@iter->iter_value()),
    @iter->iter_advance()
```

```
  *]
]*];

StructProc
is
a
strange
language
stdOut
```

Note:

- Iterators are in class `Iterator`.
- `dataStruct_iter()` returns an iterator at the beginning of a data-structure: `VectorList`, `NodeList`, `Bag` or `Map`.
- `iter_isAtEnd()` returns `true` when the iterator is at the end, or `false` otherwise.
- `iter_value()` returns the next value in the data-structure. For maps, it returns the mapped-to value. For maps use `iter_key()` to get the key of the current pair.
- `iter_advance()` advances the iterator to the next item.
- `iter_reset()` resets the iterator to the beginning of the data-structure.

C) The new operator

Use the `new` operator to dynamically create a new instance of class, that is, at run-time. It uses the same constructor as anonymous implicit construction with `^` at compile time.

Run-time? Compile-time? You can see the difference between `^VectorList[* *]` and `new VectorList[* *]` below:

Code	Output

`^Do`
`[*[`
`  ^VarDecl[* @i, Integer, 1 *],`
`  ^For`
`  [*`
`    null,`
`    @i <= 10,`
`    stdOut->`
`      printLn`
`        (^VectorList[* *]->dataStruct_insert(@i)),`
`    @i := @i + 1`
`  *]`
`]*];`
	[1]
	[1,2]
	[1,2,3]
	[1,2,3,4]
	[1,2,3,4,5]
	[1,2,3,4,5,6]
	[1,2,3,4,5,6,7]
	[1,2,3,4,5,6,7,8]
	[1,2,3,4,5,6,7,8,9]
	[1,2,3,4,5,6,7,8,9,10]
	stdOut

`^Do`
`[*[`
`  ^VarDecl[* @i, Integer, 1 *],`
`  ^For`
`  [*`
`    null,`
	`[1]`
	`[2]`
	`[3]`
	`[4]`
	`[5]`
	`[6]`

Code	Output

`    @i <= 10,`
`    stdOut->`
`      printLn`
`        (new VectorList[* *]->dataStruct_insert(@i)),`
`    @i := @i + 1`
`  *]`
`]*];`
	`[7]`
	`[8]`
	`[9]`
	`[10]`
	`stdOut`

The first code snippet uses anonymous implicit construction with `^`. One `VectorList` was created before the code ever ran. All items were put into it. The second code snippet uses the `new` operator. A new `VectorList` is created every time the loop runs. Thus, every new list only has one item.

Unlike Lisp, StructProc does not yet support the dynamic creation of new code. Therefore `new Do`, `new For`, `new VarDecl`, *etc* are all prohibited.

X. The Ontology and Nomenclature