1. About StructProc, and About this Document.

StructProc 9a is the internal knowledge representation language of the Scienceomatic 9a: an environment for scientific reasoning, visualization, discovery, education, *etc*. Its design borrows from several computer languages including C++, Prolog, Java and Ruby. It also borrows from Lisp, including morphology of the name: Lisp = *LIS*t *P*rocessing; StructProc = *STRUCT*ure *PROC*essing. More succinctly it is named SP9a.

Although StructProc9a borrows from several computer languages, it has several features that combine to make it uniquely for science:

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`.

2. How to start StructProc, and how to leave.

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

```
$ sp9a
```

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 StructProc 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
$
```

StructProc has several options that you may give on the command line to tell how to run it. Besides running it 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:

```
$ ./sp9a --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://127.0.0.1:8080` into your browser. Like the introductory message says, press `Ctrl-C` in command-line window to stop the HTTP server.

Only one StructProc9 server process 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.

To run it in batch mode where all input (after libraries are read) comes from file `input.som9`, all output should go to file `output.txt`, and the normal printing of the language and library files should not be printed, start StructProc with:

3. Non-Annotated Boolean, String and Numeric Constants

StrucProc has 3 classes of non-annotated data: booleans, strings and numbers. Booleans (class `Boolean`) have two legal values: `false` and `true`. String constants are composed of Unicode characters and are delimited by double quote characters. Class `Number` is sub-divided into four sub-classes: `Integer`, `Rational`, `Real` and `Complex`. The term "non-annotated" means that without meta-data[1], thus the non-annotated numbers lack dimensionality, units, subject, *etc*.

One important note: in SP9a methods are run with the "arrow" operator `->`. This syntax is taken from C++ for running a method given a ***pointer*** to an instance, but is different from the period operator used in Java, and is different from the period operator used in C++ for running a method given an instance directly. The reason for the arrow is because it allows the language to run methods on instances of `Integer`. So this is supported:

```
SP9a1 :) [ 3]  1->toString() ++con 2->toString();
12
SP9a1 :) [ 4]  1 + 2;
3
```

The former runs the `toString()` methods on both `Integer` instances `1` and `2`, and then concatenates the results to form the string `"12"`. The latter just adds both to compute `Integer` instance `3`.

Class `Boolean` of course has two values: `false` and `true`. The supported operators on `Boolean` values are `and`, `or`, equality and inequality. They are listed in order of decreasing precedence in the table below:

Operator	Associativity
`!`	N/A
`==ref  !=ref`	left-to-right
`and`	left-to-right
`or`	left-to-right

The operator `==ref` returns `true` if the two operand values on either side of it ***refer*** to the same fundamental thing. The "`==`" part is taken from the C programming language. The operator `!=ref` returns `true` if the two operand values on either side of it do ***not*** refer to the same fundamental thing. Again, the "`!=`" part is taken from the C programming language.

The names of operators `and` and `or` are taken from Pascal. However, like in C, `and` and `or` both "short-circuit". That is, operator `and` does not compute the right-hand-side operand if the left-hand-side operand computes to `false`: it just returns `false`. Likewise, operator `or` does not compute the right-hand-side operand if the left-hand-side operand computes to `true`: it just returns `true`.

String constants are composed of unicode characters and are delimited by double quote characters.

```
SP9a1 :) [ 1]  "Hello world!";
Hello world!
SP9a1 :) [ 1]
```

1 For the computer scientists they are implemented as immutable singletons.

String constants may use the same backslash-escape sequence characters as defined in C:

newline	NL (LF)	\n	audible alert	BEL	\a
horizontal tab	HT	\t	backslash	\	\\
vertical tab	VT	\v	question mark	?	\?
backspace	BS	\b	single quote	'	\'
carriage return	CR	\r	double quote	"	\"
formfeed	FF	\f			

```
SP9a1 :) [ 1]  "\n\tHello\n";

        Hello
```

Strings may be concatenated with the `++con` operator.

```
SP9a1 :) [ 2]  "Hello " ++con "there!";
Hello there!
```

There are also two string "uncatenation"[2] operations. Both use two operators. The `--unc` operator goes between the two strings. The `~` operator may go in *front* of the second string meaning the second string's occurrence should be removed from the *end* of the first string.

```
SP9a1 :) [ 3]  "redoing" --unc ~"ing";
redo
```

The `~` operator may also go in *behind* of the second string meaning the second string's occurrence should be removed from the *beginning* of the first string.

```
SP9a1 :) [ 4]  "redoing" --unc "re"~;
doing
```

In either case if the second string does not occur in the first string then the result just is the first string unmodified.

```
SP9a1 :) [ 5]  "redoing" --unc ~"ING";
redoing
SP9a1 :) [ 6]  "redoing" --unc "RE"~;
redoing
```

We will get to methods later, but while we are on the topic of `String` instances, it is natural to introduce these methods on them:

2 "Concatenate" is from the Latin: "con-" (with) "caten(a)" (chain). By similar construction "uncatenate" is meant to mean "unchaining", esp. of strings.

String method:	Purpose:
`string_length()`	Returns the length of the string.
`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`.

```
sp8b :) [12] "12345"->string_length();
5
sp8b :) [13] "1234"->string_length();
4
sp8b :) [14] "123"->string_length();
3
sp8b :) [15] "12"->string_length();
2
sp8b :) [16] "1"->string_length();
1
sp8b :) [17] ""->string_length();
0
sp8b :) [18] "hello"->string_capitalize();
Hello
sp8b :) [19] ""->string_capitalize();

sp8b :) [20] "123"->string_capitalize();
123
sp8b :) [21] "HELLO"->string_capitalize();
HELLO
sp8b :) [22] "redoing"->string_hasPrefix("redo");
true
sp8b :) [23] "redoing"->string_hasPrefix("re");
true
sp8b :) [24] "redoing"->string_hasPrefix("r");
true
sp8b :) [25] "redoing"->string_hasPrefix("Redo");
false
sp8b :) [26] "redoing"->string_hasPrefix("Re");
false
```

```
sp8b :) [27] "redoing"->string_hasPrefix("R");
false
sp8b :) [28] "redoing"->string_prefix(4);
redo
sp8b :) [29] "redoing"->string_prefix(3);
red
sp8b :) [30] "redoing"->string_prefix(2);
re
sp8b :) [31] "redoing"->string_prefix(1);
r
sp8b :) [32] "redoing"->string_prefix(0)->string_length();
0
sp8b :) [33] "redoing"->string_prefix(1000000);
redoing
sp8b :) [34] "redoing"->string_hasSuffix("ing");
true
sp8b :) [35] "redoing"->string_hasSuffix("ng");
true
sp8b :) [36] "redoing"->string_hasSuffix("g");
true
sp8b :) [37] "redoing"->string_hasSuffix("ion");
false
sp8b :) [38] "redoing"->string_hasSuffix("on");
false
sp8b :) [39] "redoing"->string_hasSuffix("n");
false
sp8b :) [40] "redoing"->string_suffix(3);
ing
sp8b :) [41] "redoing"->string_suffix(2);
ng
sp8b :) [42] "redoing"->string_suffix(1);
g
sp8b :) [43] "redoing"->string_suffix(0)->string_length();
0
sp8b :) [44] "redoing"->string_suffix(1000000);
redoing
sp8b :) [45] "redoing"->string_substring(0,2);
re
sp8b :) [46] "redoing"->string_substring(2,2);
do
sp8b :) [47] "redoing"->string_substring(4,3);
ing
sp8b :) [48] "redoing"->string_substring(10,10)->string_length();
0
```

There are 4 subclasses of Number: `Rational` and its subclass `Integer`, and `Complex` and its

subclass `Real`.[3] Integers are denoted by just typing them. Rationals are denoted as two integers separated by the backslash. You may type them is as any integer over any other integer but they are stored such that the denominator is always positive and all common primes are removed:

```
SP9a1 :) [ 1]  5\10;
1\2
SP9a1 :) [ 1]  5\-10;
-1\2
SP9a1 :) [ 2]  -5\10;
-1\2
SP9a1 :) [ 3]  -5\-10;
1\2
```

The methods `rational_numerator()` and `rational_denominator()` are defined on instances of `Rational`. They return the numerator and denominator of the number respectively. For instances of `Integer`, `rational_numerator()` always returns that integer itself while `rational_denominator()` always returns `1`:

```
SP9a1 :) [ 6]  1\2->rational_numerator();
1
SP9a1 :) [ 7]  1\2->rational_denominator();
2
SP9a1 :) [ 8]  4->rational_numerator();
4
SP9a1 :) [ 9]  4->rational_denominator();
1
```

Reals may either be given with a decimal point, or in floating point exponent notation:

```
SP9a1 :) [ 2]  10.0;
10.
SP9a1 :) [ 3]  1e+1;
10.
```

Complex numbers have an "`i`" postfixing their imaginary component. Just "`i`" by itself denotes one times the square root of `-1`. Complex numbers whose real and imaginary components are non-zero are wrapped in parentheses:

```
SP9a1 :) [ 4]  i;
i
SP9a1 :) [ 5]  2 + 2i;
(2.+2.i)
```

The following unary functions are defined on instances of both `Real` and `Complex`. (For these functions both `Integer` and `Rational` are cast to `Real`, so they may use used too.)

Name:	Purpose:
`sin(x)`	Sine of x radians

3 Yes, technically Complex ⊇ Real ⊇ Rational ⊇ Integer, but the internal representation of Complex and Real is both as floating points, while that of Rational and Integer is both as integers. The class hierarchy reflects these differences. Integers are just rationals whose denominator is `1`. Reals are just complex numbers whose imaginary component is just `0.0`.

Name:	Purpose:
`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
`log2(x)`	Logarithm base 2
`log10(x)`	Logarithm base 10

Non-annotated numbers support the typical operations. They are listed in order of decreasing precedence in the table below:

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` instances. 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 :) [ 1]  2 ==num 2.0;
true
SP9a1 :) [ 1]  2 ==ref 2.0;
false
SP9a1 :) [ 2]  3\2 !=num 1.5;
false
SP9a1 :) [ 3]  3\2 !=ref 1.5;
true
```

Later we will see three algebraic operators: `<<` (is-much-lesser-than), `>>` (is-much-greater-than) and `=` (equals). These do compare their left-hand and right-hand sides, nor do they return `Boolean` instances. Instead, they are used to defined algebraic relationships among expressions.

4. Basic Knowledge Assertion

An ***identity*** is some thing that the Scienceomatic knows about. It could be a name (like `Idea`) or a number (like `42`). For example, `42` is the identity of the corresponding rational integer.

One fundamental distinction among identities is that some are used as ***instances*** and others as ***classes***. Instances represent specific instantiations of classes, and may belong to one or more classes. For example, `41`, `42`, and `43` are all *instances* of the *class* of integers: `Integer`. Classes may subsume more specific classes. For example, `Integer` is a subclass of `Rational`, and Rational is a subclass of the `Number`.

StructProc9a is a pure object-oriented language. Everything that is representable ultimately an instance of the class `Idea`. Important subclasses of `Idea` include:

`Class`, an immediate subclass of `Idea`. All classes are *instances* of `Class`.

`Nonaffiliated`, an immediate subclass of `Idea`. `Nonaffiliated` is the class into which are placed classes and instances when their position in the ontology has not yet been given by the user. Such "homeless" classes are made subclasses of `Nonaffiliated`, and instances are made instances of `Nonaffiliated`. As soon as the user gives their proper position in the ontology they are removed from `Nonaffiliated`. One may not explicit assert any instance or subclass of `Nonaffiliated`, it is meant as a temporary way station.

`Attribute`, an immediate subclass of `Idea`. `Attribute` is the class of all attributes, Identity instances that represent properties of other Identity instances. Examples of attributes (*i.e.* instances of `Attribute`) include `attrsDomainA`, which maps attributes to their domains (a representation of the class they describe), and `attrsRangeA`, which maps attributes to their ranges (a representation of the class in which their values lie).

`ProgramObject`, an immediate subclass of `Idea`. `ProgramObject` is the class of all programming constructs of which StructProc9a may be made self-aware. Important immediate subclasses of `ProgramObject` include `Iterator` (the class of all iterators), `DataStructure` (the immediate superclass of `List`, `Bag` and `Map`) and `IOStream` (the class of all input/output streams).

`EmpiricalEntity`, an immediate subclass of `Idea`. `EmpiricalEntity` is the class of all things in the "real-world" of which StructProc9 should be aware. Examples include the local galactic group, the Earth, hummingbirds, and glucose molecules.

`Value`, an immediate subclass of `Idea`. `Value` is the immediate superclass of `AnnotatedValue` and `NonannotatedValue`.

`AnnotatedValue`, an immediate subclass of `Value`. Annotated values exist to represent values annotated with meta-data like dimensions, units, and the subject and attribute to which the value pertains.

`NonannotatedValue`, an immediate subclass of `Value`. Subclasses of `NonannotatedValue` include `Boolean`, `String` and `Number`. Non-annotated values were discussed in Chapter 3.

`Number`, an immediate subclass of `NonannotatedValue`. Number serves as the immediate superclass of both `Rational` and `Complex`.

`Rational`, an immediate subclass of `Number`. `Rational` is the immediate class of all representable rational numbers: rational numbers with numerators between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 inclusive with denominators between 2 and 18,446,744,073,709,551,615 inclusive after all common prime integers have been removed

from both numerator and denominator. All such representable rational numbers are instances of `Rational`. `Rational` is also the immediate superclass of `Integer`.

`Integer`, an immediate subclass of `Rational`. `Integer` is the immediate class of all representable integers: rationals whose numerators are between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 inclusive, and whose denominator is 1. All such representable integers are instances of `Integer`. (Note that `Integer` is a subclass of `Rational`, and not the other way around. Having `Rational` as a subclass of `Integer` would have the advantage of saving memory in a traditional object-oriented language because one need not store the denominator for integers. However, it is not mathematically sound. Mathematically, all integers are rationals, but not all rationals are integers.)

`Complex`, an immediate subclass of `Number`. `Complex` is the immediate class of all representable complex numbers: complex numbers with imaginary component other than 0, and whose real and imaginary components are both representable with IEEE 8-byte double floating point numbers. All such representable complex numbers are instances of `Complex`. `Complex` is also the immediate superclass of `Real`.

`Real`, an immediate subclass of `Complex`. `Real` is the immediate class of all representable real numbers: numbers whose imaginary component is 0, and whose real component is representable with IEEE 8-byte double floating point numbers. All such representable real numbers are instances of `Real`. (Note that `Real` is a subclass of `Complex`, and not the other way around. Having `Complex` as a subclass of `Real` would have the advantage of saving memory in a traditional object-oriented language because one need not store the complex component for reals. However, it is not mathematically sound. Mathematically, all reals are complex, but not all complex are real.)

`String`, an immediate subclass of `Value`. `String` is the class to which all strings belong.

`Boolean`, an immediate subclass of `Value`. `Boolean` is the class to which `true` and `false` belong.

One may, of course, define one's own classes and instances. Define a class with:

```
class MyClass
{*
*};
```

Notice the keyword `class` at the beginning and the semicolon at the end. We did not say where in the ontology `MyClass` should go, so while `MyClass` is an ***instance*** of `Class`, it is a ***subclass*** of `Nonaffiliated`.

Define an instance of `MyClass` with the following:

```
myInstance
{*
  instanceOf->assertZ(MyClass)
*};
```

Again, notice the semicolon at the end. Here, `instanceOf` is an attribute mapping `myInstance` to the class of which it is a member: `MyClass`. More will be said about asserting knowledge below.

It is the culture of StructProc9a to begin instance names with a lowercased character (*e.g.* `theKnowledgeBase`) and class names with an uppercased character (*e.g.* `ProgramObject`).

From there, both use “camel-back” morphology (capital letters marking the beginning of words in terms with minimal use of underscores) is recommended. (StructProc9a also can handle arbitrary names: names that begin with digits, contain spaces, use non-ASCII characters, *etc*. To do so, wrap the name in grave (or “back-tick”) characters: (`). An example is `2001: A Space Odyssey`.)

The properties of the Identity being defined go between the {* and *} lexemes. A ***property*** is a (attribute, value) pair. For example,

```
instanceOf->assertZ(MyClass)
```

in the definition of myInstance above defines the triplet (instanceOf, MyClass, default justification) as a property of myInstance. The attribute is instanceOf, the value is MyClass.

Identities may have multiple values for the same attribute. The values of a given attribute are kept in a list. If no properties (attribute, value pairs) have been given for an attribute then that attribute's list is empty.

Properties may be given to a node with the following methods:

- attribute->assertZ(value) adds value to the end of the list for attribute attribute,
- attribute->assertA(value) adds value to the beginning of the list for attribute attribute,
- attribute->assert(value) erases the list for attribute attribute and makes value the sole member of the list.

Properties defined in this way are said to be *locally-asserted* because the property is directly stated with the identity that it modifies. The term **assertX()** does not exist in the language, but is used to refer to **assert()**, **assertA()** and **assertZ()** collectively.

<MUST REVISE>

To see the node definition of an identity (*e.g.* theKnowledgeBase) say:

```
SP9a1 :) [1] stdOut->describeLn(theKnowledgeBase);
theKnowledgeBase
{*
  instanceOf->assertZ(NatLangNounTerm),
  ^TermsTextA[*toEnglish*]->assertZ("knowledge base")
*};
stdOut
```

stdOut is the identity for the main output file, oftentimes the user's screen. We are running the describeLn(identity) method of output files, which causes the output stream to print the definition of identity to itself, followed by a newline character. Finally, the describeLn() method returns the file to which the output was sent; this accounts for the ending stdOut in the output above.

The actual definition outputted is similar to what was described above: name, beginning {*, list of properties, enclosing *}, all separated by a comma (,). The only difference should be one or more ^TermsTextA[*..*] attribute properties. More will be said about anonymous instances (^TermsTextA) in chapter 8, and about implicit construction ([*..*]) in chapter 7.

There is also a `describe()` method that omits the newline character; its output is generally less attractive.

```
SP9a1 :) [2] stdOut->describe(theKnowledgeBase);
theKnowledgeBase
{*
  instanceOf->assertZ(NatLangNounTerm),
  ^TermsTextA[*toEnglish*]->assertZ("knowledge base")
*};
stdOut
```

Classes have unique attributes by virtue of being classes. The `isA` attribute tells of another class (the "superclass") that has the given class as a subclass, just as the `instanceOf` attribute tells of a class that has the given instance as an instance of itself. In the partial definition below the class `NatLangNounTerm` is noted as a subclass of both `NatLangTerm` and `NatLangNounPhrase` with the `isA` attribute.

```
NatLangNounTerm
{*
  natLangNounPhrasesNounA->subAssertZ(theSubject),
  nounPhrasesPluralityA->subAssertZ(singularPlurality),
  nounPhrasesPersonA->subAssertZ(thirdPerson),

  isA->assertZ(NatLangTerm),
  isA->assertZ(NatLangNounPhrase)
*};
```

</MUST REVISE>

However, beyond specifying properties for themselves, classes may also state properties that are inherited by instances of themselves. Such values are also managed with lists:

- `attribute->subAssertZ(value)` adds value `value` to the end of attribute `attribute`'s list,
- `attribute->subAssertA(value)` adds value `value` to the beginning of attribute `attribute`'s list,
- `attribute->subAssert(value)` erases attribute `attribute`'s list and makes value `value` its sole member.

<MUST REVISE>
In the example above `theKnowledgeBase` (as an instance of `NatLangNounTerm`) will inherit the property that attribute `nounPhrasesPluralityA` has value `singularPlurality`, and the property that attribute `nounPhrasesPersonA` has value `thirdPerson`.
</MUST REVISE>

Attributes `instanceOf` and `isA` are root-wards pointing attributes: they build the ontology by pointing towards more general classes. There are also instances-wards pointing attributes that point from general classes to more specific classes, and to instances. Attribute `hasInstance` is the inverse

of attribute `instanceOf`. If instance `instance` asserts that it is an `instanceOf` class `class`, then StructProc9a will automatically assert that class `class` has `hasInstance` value `instance`. Similarly, attribute `hasSubclass` is the inverse of attribute `isA`. You can see this by looking at the definition of class `NatLangNounTerm`:

<MUST REVISE>

```
SP9a1 :) [3] stdOut->describeLn(NatLangNounTerm);
NatLangNounTerm
{*
  natLangNounPhrasesNounA->subAssertZ(theSubject),
  nounPhrasesPluralityA->subAssertZ(singularPlurality),
  nounPhrasesPersonA->subAssertZ(thirdPerson),

  isA->assertZ(NatLangTerm),
  isA->assertZ(NatLangNounPhrase),
  hasSubclass->assertZ(NatLangDynamicallyConstructedNounTerm),
  hasInstance->assertZ(theOuterSubject),
  hasInstance->assertZ(theLastRequestedComputation)
  . . .
*};
stdOut
```

Attributes `hasSubclass` and `hasInstance` are listed by the `describe()` and the `describeLn()` methods for completeness. They should ***not*** be directly asserted, however. They are automatically asserted whenever attributes `isA` (in a subclass) and `instanceOf` (in an instance) are asserted, respectively.

</MUST REVISE>

Ideally asserted properties have their values already defined. However, there are times when this is not possible (*e.g*. for circular references). In such cases one should ***declare*** an Identity before it is defined. (Declaration of instances in StructProc9a is similar in spirit to declaration of symbols in C/C++.) To declare something use the syntax:

```
instance in ClassOfInstance
```

Here, *`instance`* may itself be a class, just have *`ClassOfInstance`* be the class `Class`, because all classes are instances of `Class`.

```
// PURPOSE: To represent the set of attributes of Widgets:
class WidgetA
{*
  isA->assertZ(EmpiricalObjectA),
  // WidgetA is a subclass of the attributes of empirical objects

  attrsDomainA->subAssert(Widget in Class)
  // We assert that the domain of instances of WidgetA is Widget,
  // even though Widget has yet to be defined.
*};
...
```

```
// Finally we define what Widgets are:
// PURPOSE: To represent the set of Widgets.
class Widget
{*
  isA->assertZ(EmpiricalObject)
*};
```

Attempting to define a class for an Identity different from the class to which it has been declared will result in an error.

(example)

Attempting to re-declare a class for an Identity different from the class to which it was originally declared will also result in an error.

(example)

One may assert declared Identities as values in properties. However, because they are not yet defined one may not query over them. Furthermore, even though one has stated their class, if one iterates over the instances of that class, they will not show up. They will only show up after they have been defined.

If there is anything that declared but not defined then the knowledge base is in an ***unfinished*** state. It cannot be saved until all declared Identity instances have been defined.

The subject `thisSE` (for "***this*** ***S****cienceomatic* ***E****nvironment*") represents the currently running system. One changes the behavior of the system by asserting properties to it. For example, the system will tell you the assembly language that its virtual machine will run by setting its attribute `spEnvsShouldDisplayPreAssemPostOptA` to `true`:

```
SP9a1 :) [ 1]  thisSE->assert(spEnvsShouldDisplayPreAssemPostOptA,true);
thisSE
SP9a1 :) [ 2]  true and false;
   loadPoin     %0x0, const(0x0,0x5) # true
   ifFalseGotoPoin %0x0, label_FFFFFFFF
   loadPoin     %0x0, const(0x0,0x4) # false
   copyPoin     %0x0, %0x0
   stopWSuccessPoin


false
```

To turn this off, just assert the value false for the same attribute:

```
SP9a1 :) [ 3]  thisSE->assert(spEnvsShouldDisplayPreAssemPostOptA,false);
   loadPoin     %0x0, const(0x0,0x1C)   # thisSE
   loadPoin     %0x1, const(0x0,0x37)   # spEnvsShouldDisplayPreAssemPostOptA
   loadPoin     %0x2, const(0x0,0x4) # false
   assertPoin      %0x0, %0x1, %0x2, %0x2
   copyPoin     %0x0, %0x0
   stopWSuccessPoin


thisSE
SP9a1 :) [ 4]  true and false;
false
```

By default, the system starts out at its lowest level of optimization. However, this may be changed by asserting attribute `spEnvsOptimizationDegreeA`. Its legal values are `optimizeDegreeLowest`, `optimizeDegree2ndLowest`, `optimizeDegree2ndHighest` and `optimizeDegreeHighest`. We can see the affect of the optimization by looking at the generated assembly language code. At the lowest optimization degree the expression (true and false) is computed with the following assembly language snippet:

```
SP9a1 :) [ 5]  thisSE->assert(spEnvsOptimizationDegreeA,optimizeDegreeLowest);
thisSE
SP9a1 :) [ 6]  thisSE->assert(spEnvsShouldDisplayPreAssemPostOptA,true);
thisSE
SP9a1 :) [ 7]  true and false;
   loadPoin     %0x0, const(0x0,0x5) # true
   ifFalseGotoPoin %0x0, label_FFFFFFFF
   loadPoin     %0x0, const(0x0,0x4) # false
   copyPoin     %0x0, %0x0
   stopWSuccessPoin


false
SP9a1 :) [ 8]  thisSE->assert(spEnvsShouldDisplayPreAssemPostOptA,false);
   . . .
```

However, even at the 2nd lowest level of optimization, the expression is computed at compile-time. So the resulting run-time assembly language is:

```
SP9a1 :) [ 9]  thisSE->assert(spEnvsOptimizationDegreeA,optimizeDegree2ndLowest);
thisSE
SP9a1 :) [ 10]  thisSE->assert(spEnvsShouldDisplayPreAssemPostOptA,true);
thisSE
SP9a1 :) [ 11]  true and false;
   loadPoin     %0x0, const(0x0,0x4) # false
   stopWSuccessPoin


false
SP9a1 :) [ 12]  thisSE->assert(spEnvsShouldDisplayPreAssemPostOptA,false);
   . . .
```

zzz

5. Basic Querying

5.1 `get()`, `localGet()`, `subGet()` and `theSubject`:

Once asserted, knowledge may be queried. The most general way to get the first of attribute `attr` is with the `get(attr)` method. It will retrieve both locally-asserted and inherited property values.

6. Intermediate Knowledge Assertion.

6.1 Continuing an Identity's Definition.

In StructProc, unless we explicitly say otherwise, we may add more knowledge to a node. For example, even though `theKnowledgeBase` already has a definition like:

6.2 Assigning nicknames

Units often have abbreviations: “m” for “meters”, “kg” for “kilograms”, *etc*. StructProc9A can assign nicknames to Identities by asserting a list of strings of nicknames to the `nicknameListA` attribute.

For example, for meters we may state:

```
meters
{*
  nicknameListA->assertZ(["m"])
*};
```

Now, *assuming “m” is not the name of an existing Identity*, “m” will actually connote “meters”.

```
SP9a1 :) [1] m;
meters
```

The argument to the `nicknameListA` assertion must be a list of strings.

You may overwrite nicknames with new nicknames, but you will be warned each time you do.

```
SP9a1 :) [2] moles {* nicknameListA->assert(["m"]) *};
WARNING: Nickname "m" used to map to meters, now to map to moles
SP9a1 :) [3] m;
moles
SP9a1 :) [4] meters {* nicknameListA->assert(["m"]) *};
WARNING: Nickname "m" used to map to moles, now to map to meters
SP9a1 :) [5] m;
meters
```

6.3 Classes with Automatically-Named Instances:

StructProc9 allows the user to specify that new instances of some classes be named in a systematic fashion by the system. This allows users to know in advance what the name of the next newly-created instance of a class will be.

The purpose of this feature is to support World Wide Web protocols. World Wide Web protocols favor communications that are “idempotent”: where the effect of doing them once is the same as doing them several times. If activity pages were anonymous, then doing several uploads of the definition of a single new page to a StructProc9 server via HTTP method PUT would result in several pages. This is *not* idempotent. However, if the pages were *named*, then uploading the same named page several times would be the same as uploading it once. This is idempotent.

This can only be done if the name of the next class instance can be established in advance. For

classes, the method `getNextAutomaticInstanceName()` either returns the name of the next instance of that class:

```
SP9a1 :) [ 1] UserActivity->getNextAutomaticInstanceName();
__UserActivity__0__
SP9a1 :) [ 1] ^UserActivity;
__UserActivity__0__
SP9a1 :) [ 2] UserActivity->getNextAutomaticInstanceName();
__UserActivity__1__
```

or `null` if the class is not noted as having automatically-named instances.

```
SP9a1 :) [ 3] Boolean->getNextAutomaticInstanceName();
null
```

To state that a class has automatically-generated instances, `assertX` that its value for attribute `areInstancesAutomaticallyNamedA` is `true.`

```
class  UserActivity
{*
  isA->assert(ProgramObject),

  areInstancesAutomaticallyNamedA->assert(true)
*};
```

7. Constructors and Implicit Construction

7.1 Basic Constructors

Supposed we wanted to inform our knowledge base about the results of simultaneously flipping five coins: a 1-cent piece, a 5-cent piece, a 10-cent piece, a 25-cent piece, and a 50-cent piece. We could define "5 coin flip" as being process:

```
SP9a1 :) [ 1] class Process5CoinFlip
{*
  isA->assertZ(EmpiricalEntity)
*};
Process5CoinFlip
```

Above we define `Process5CoinFlip` as a subclass of `EmpiricalEntity`. Because it is a class, we follow the convention of beginning it with a capital letter.

Next, we define attributes for "was the 1 cent piece heads", "was the 5 cent piece heads", "was the 10 cent piece heads", "was the 25 cent piece heads" and , "was the 50 cent piece heads" as Boolean attributes of empirically existing things (rather than, say human-defined things, like meters and kilograms):

```
SP9a1 :) [ 2] was1CentPieceHeadsA
{*
  instanceOf->assertZ(BooleanEmpiricalA)
*};
was1CentPieceHeadsA
SP9a1 :) [ 3] was5CentPieceHeadsA
{*
  instanceOf->assertZ(BooleanEmpiricalA)
*};
was5CentPieceHeadsA
SP9a1 :) [ 4] was10CentPieceHeadsA
{*
  instanceOf->assertZ(BooleanEmpiricalA)
*};
was10CentPieceHeadsA
SP9a1 :) [ 5] was25CentPieceHeadsA
{*
  instanceOf->assertZ(BooleanEmpiricalA)
*};
was25CentPieceHeadsA
SP9a1 :) [ 6] was50CentPieceHeadsA
{*
  instanceOf->assertZ(BooleanEmpiricalA)
*};
was50CentPieceHeadsA
```

Defining the identities above must be done before the coin-flipping data makes sense. However, the

only method we have seen for giving the coin-flipping data would be verbose and tedious[4]:

```
SP9a1 :) [ 7] first5CoinFlip
{*
  instanceOf->assertZ(Process5CoinFlip),

  was1CentPieceHeadsA ->assertZ(true),
  was5CentPieceHeadsA ->assertZ(false),
  was10CentPieceHeadsA->assertZ(true),
  was25CentPieceHeadsA->assertZ(true),
  was50CentPieceHeadsA->assertZ(true)
*};
first5CoinFlip
SP9a1 :) [ 8] second5CoinFlip
{*
  instanceOf->assertZ(Process5CoinFlip),

  was1CentPieceHeadsA ->assertZ(true),
  was5CentPieceHeadsA ->assertZ(false),
  was10CentPieceHeadsA->assertZ(true),
  was25CentPieceHeadsA->assertZ(false),
  was50CentPieceHeadsA->assertZ(false)
*};
second5CoinFlip
SP9a1 :) [ 8] third5CoinFlip
{*
  instanceOf->assertZ(Process5CoinFlip),

  was1CentPieceHeadsA ->assertZ(true),
  was5CentPieceHeadsA ->assertZ(false),
  was10CentPieceHeadsA->assertZ(false),
  was25CentPieceHeadsA->assertZ(false),
  was50CentPieceHeadsA->assertZ(false)
*};
third5CoinFlip
```

Fortunately, there is an alternative: ***implicit construction***. Implicit construction allows us to list the values of attributes. StructProc assigns values to their corresponding attributes in the order in which both attributes and values are listed.

Let us add a constructor to `Process5CoinFlip's` definition:

```
SP9a1 :) [ 9] class Process5CoinFlip
{*
  implicitConstructorA->
 subAssert
 (^Constructor
   [*
```

4 Thank you to Random.org for the data.

```
      [[ was1CentPieceHeadsA, Boolean],
       [ was5CentPieceHeadsA, Boolean],
       [ was10CentPieceHeadsA,Boolean],
       [ was25CentPieceHeadsA,Boolean],
       [ was50CentPieceHeadsA,Boolean]
      ]
    *]
 )
*};
Process5CoinFlip
```

Above, we give instances of `Process5CoinFlip` (rather than `Process5CoinFlip` itself) a value for attribute `implicitConstructorA` with `subAssert`. That value is:

```
^Constructor
   [ *
     [[ was1CentPieceHeadsA, Boolean],
      [ was5CentPieceHeadsA, Boolean],
      [ was10CentPieceHeadsA,Boolean],
      [ was25CentPieceHeadsA,Boolean],
      [ was50CentPieceHeadsA,Boolean]
     ]
   *]
```

We give `^Constructor` one value[5]: the list of binary lists:

```
      [[ was1CentPieceHeadsA, Boolean],
       [ was5CentPieceHeadsA, Boolean],
       [ was10CentPieceHeadsA,Boolean],
       [ was25CentPieceHeadsA,Boolean],
       [ was50CentPieceHeadsA,Boolean]
      ]
```

Each binary list tells both an attribute and a domain for that attribute. Thus, above we list all 5 attributes and tell that all of the domains of their values is `Boolean` (that is, the values must be either `true` or `false`).

7.2 Implicit Construction of Named Instances

Now that the constructor is defined we can use it to create three more coin-flipping events:

```
SP9a1 :) [ 10] fourth5CoinFlip[*Process5CoinFlip|true,true,false,
false,false*];
fourth5CoinFlip
SP9a1 :) [ 11] fifth5CoinFlip[*Process5CoinFlip|true,true,false,
true,true*];
fifth5CoinFlip
```

5 What does `^Constructor[* .. *]` mean? Please wait until chapter 7!

```
SP9a1 :) [12] sixth5CoinFlip[*Process5CoinFlip|false,true,false,
true,false*];
sixth5CoinFlip
```

Note the syntax:

- the identity name,
- a beginning `[*`,
- the class name,
- a vertical bar (`|`),
- the values in order separated by commas,
- an ending `*]`.

The final semicolon (`;`) makes signifies that it is a complete StructProc expression.

This creates three more `Process5CoinFlip` instances just as if we created then with explicit construction (`{* .. *}`), only easier. We can see this by looking at them:

```
<must revise>
SP9a1 :) [13] stdOut->describeLn(first5CoinFlip);
first5CoinFlip
{*
  instanceOf->assertZ(Process5CoinFlip),
  was1CentPieceHeadsA->assertZ(true),
  was5CentPieceHeadsA->assertZ(false),
  was10CentPieceHeadsA->assertZ(true),
  was25CentPieceHeadsA->assertZ(true),
  was50CentPieceHeadsA->assertZ(true)
*};
stdOut
SP9a1 :) [14] stdOut->describeLn(fourth5CoinFlip);
fourth5CoinFlip
{*
  instanceOf->assertZ(Process5CoinFlip),
  was1CentPieceHeadsA->assertZ(true),
  was5CentPieceHeadsA->assertZ(true),
  was10CentPieceHeadsA->assertZ(false),
  was25CentPieceHeadsA->assertZ(false),
  was50CentPieceHeadsA->assertZ(false)
*};
stdOut
</must revise>
```

7.3 Implicit Construction with Default Values

Implicit construction may also be used to define default values. Let us say we suspect the 50 cent piece to be slightly biased to give heads over tails. We could redefine the inherited constructor as:

```
SP9a1 :) [15] class Process5CoinFlip
{*
  implicitConstructorA->
 subAssert
 (^Constructor
```

```
    [*
      [[was1CentPieceHeadsA, Boolean],
       [was5CentPieceHeadsA, Boolean],
       [was10CentPieceHeadsA,Boolean],
       [was25CentPieceHeadsA,Boolean],
       [was50CentPieceHeadsA,Boolean]
      ],
      [[was50CentPieceHeadsA,false]
      ]
    *]
 )
*};
Process5CoinFlip
```

Here, the call to `^Constructor[*..*]` is given two lists. The first is the same as before. The second, however, is another list of binary lists:

```
      [[was50CentPieceHeadsA,false]
      ]
```

The sole binary list tells an attribute (`was50CentPieceHeads`) and its default value (`false`). This value will be given to implicitly constructed instances of `Process5CoinFlip` before the listed values are. This allows the listed values to overwrite default values. Thus, if the seventh flip's 50 cent piece comes out tails, we may abbreviate this as:

```
SP9a1 :) [16] seventh5CoinFlip[*Process5CoinFlip|true,true,true,
true*];
seventh5CoinFlip
```

seventh5CoinFlip's property for attribute *was50CentPieceHeadsA* is `false`, even though we did not explicitly say so during its construction. We can see this both by querying the attribute and by describing the identity:

```
<must revise>
SP9a1 :) [17] seventh5CoinFlip->localGet(was50CentPieceHeadsA);
false
SP9a1 :) [18] stdOut->describeLn(seventh5CoinFlip);
seventh5CoinFlip
{*
  instanceOf->assertZ(Process5CoinFlip),
  was1CentPieceHeadsA->assertZ(true),
  was5CentPieceHeadsA->assertZ(true),
  was10CentPieceHeadsA->assertZ(true),
  was25CentPieceHeadsA->assertZ(true),
  was50CentPieceHeadsA->assertZ(false)
*};
stdOut
```

```
</must revise>
```

However, if the eighth flip's 50 cent piece comes out heads, then we can override it by explicitly giving a fifth value:

```
SP9a1 :) [ 19] eighth5CoinFlip[*Process5CoinFlip|true,true,true,
true,true*];
eighth5CoinFlip
```

eighth5CoinFlip's ***first*** property for attribute *was50CentPieceHeadsA* is `true`, as querying attests. The results obtained by describing it may, however, be unexpected:

```
<must revise>
SP9a1 :) [ 20] eighth5CoinFlip->localGet(was50CentPieceHeadsA);
true
SP9a1 :) [ 21] stdOut->describeLn(eighth5CoinFlip);
eighth5CoinFlip
{*
  instanceOf->assertZ(Process5CoinFlip),
  was1CentPieceHeadsA->assertZ(true),
  was5CentPieceHeadsA->assertZ(true),
  was10CentPieceHeadsA->assertZ(true),
  was25CentPieceHeadsA->assertZ(true),
  was50CentPieceHeadsA->assertZ(true),
  was50CentPieceHeadsA->assertZ(false)
*};
stdOut
</must revise>
```

Describing `eighth5CoinFlip` shows it has two value: the first being `true` and the second `false`.

Having two values for a single-valued attribute may seem weird, but it reflects one of StructProc's design philosophies. StructProc is for science, and scientists often want *alternatives:* the given value *and* the default value. Both are present.

7.4 Mixed Implicit and Explicit Construction

The last topic of this section is that implicit construction and explicit construction may be mixed. Say we also had an ordinary six-sided die. We could defined an attribute to map to the value of its roll:

7.5 The `toJson()` method

The method `toJson()` uses the constructor to build the string giving the Json definition of anonymous identities. The format is:

```
{ "type":"<classOfIdentity>",
  "constructorAttr0":<recursiveToJsonCallOnValue0>,
  . . .
  "constructorAttrN-1":<recursiveToJsonCallOnValueN-1>,
}
```

An example is below:

8. Anonymous and Single-Distinct Identities

8.1 Anonymous Identities

The last chapter introduced implicit constructors and implicit construction of named identities. Sometimes, however, we wish to create an identity without bothering to name it. This is commonly done for programming elements like loops and conditions (covered in the next chapter).

Unnamed identities are called ***anonymous*** identities. To make one, let us again create a class that represents the simultaneously flipping of five coins: a 1-cent piece, a 5-cent piece, a 10-cent piece, a 25-cent piece, and a 50-cent piece as we did in chapter 7, and give it a constructor:

8.1 Single-Distinct Identities

Consider `ToUnitsAttr`, a class of attributes that tell how to convert from one set of units to another:

```
kilometers
{*
  instanceOf->assertZ(DistanceUnits),
  unitsDefaultDomainA->assertA(defaultKmDomain),
  ^ToUnitsAttr[*meters*]->
        assertA(^LinearUnitsToUnitsStruct[*1\1000*])
*};
```

Here, the last assertion twice uses constructors to create instances. The attribute represents "*that this is how one converts from the subject (i.e. kilometers) to meters*". The value states "multiply it by 1/1000".

There is no problem with the constructor usage to create a `LinearUnitsToUnitsStruct` instance. Of course, one would use identical structures to converts from kilograms to grams, or kilocalories to calories. But there is no problem with creating identical `LinearUnitsToUnitsStruct` instances for those other cases.

Contrast that, however, with the constructor usage to create a `ToUnitsAttr` instance. If we got one instance of `ToUnitsAttr` when assert the knowledge (*e.g.* for converting kilometers to meters), and another instance of `ToUnitsAttr` when querying "*How to convert from kilometers to meters?*", then we would never find the previously stated property.

Single-distinct identities solve this problem. By asserting a class as "single-distinct", every time an instance is created by passing identical arguments to the constructor, the same identity is returned. Thus, they are like named identities in that they are always returned when uniquely called upon, but they are still anonymous because they do not have unique names.

Single-distinctness is a ***local*** property of the class.

```
class ToUnitsAttr
{*
  …
  isSingleDistinctA->assertZ(true)
*};
```

By asserting value `true` for class attribute `isSingleDistinctA`, any instance created thereafter will be registered at single-distinct. Constructor calls with the exact same arguments will always return

the same identity. Because of this, such identities are not allowed to change their properties. Thus, single-distinct instances are also immutable (have attribute `isImmutableA` value `true`).

9. Annotated Values

9.1 Introduction

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 *);[6]
```

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 create 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 that 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"`.

---

6 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.

Attributes of instances of `Domain` include:

`domainsDimensionA`: maps from instance of `Domain` to an instance of `Dimension`.

`domainsUnitsA`: maps from instance of `Domain` to an instance of `Units`.

`domainsDefaultAttrA`: maps from instance of `Domain` to an instance of `EmpiricalEntityA`, attributes of empirical entities.

Attributes of instances of `Dimension` include:

`isDimensionConventionallyFundamentalA`: maps from instance of `Dimension` to `true` if that dimension is conventionally considered to be fundamental, or to `false` otherwise.

`dimensionsDefaultDomainA`: maps from instance of `Dimension` to the `Domain` instance that that dimension uses by default.

`dimensionsDefaultAttrA`: maps from instance of `Dimension` to the `EmpiricalEntityA` instance that that dimension uses by default.

`compositeDimensionsBasicDimListA`: maps from a non-conventionally fundamental dimension to a `VectorList` of `CompositeDimensionsBasicDimStruct` instances that give the fundamental dimensions and powers of the non-conventionally fundamental subject. For example, the `compositeDimensionsBasicDimListA` property dimension `Force` is:

```
compositeDimensionsBasicDimListA->
assertZ
     ([ ^CompositeDimensionsBasicDimStruct[*Mass,1*],
        ^CompositeDimensionsBasicDimStruct[*Distance,1*],
        ^CompositeDimensionsBasicDimStruct[*Time,-2*]
      ]);
```

Attributes of instances of `Units` include:

`unitsDefaultDomainA`: maps from instance of `Units` to the instance of `Domain` that that unit uses by default.

`unitsDimensionA`: maps from instance of `Units` to the instance of `Dimension` that that unit use measures.

`unitsDefaultAttrA`: maps from instance of `Units` to the instance of `EmpiricalEntityA` that that unit describes by default.

`compositeUnitsBasicUnitToPowerMapA`: maps from instance of `Units` of a non-fundamental dimension to a map. This map maps, units of fundamental dimensions to their powers to tell how the subject non-fundamental unit is composed.

`nicknameListA`: maps from instance of `Units` to a `VectorList` that gives nicknames of that unit as strings.

For example, the following is the definition of `Newtons`:

```
Newtons
{*
  instanceOf->assertZ(Units),
  unitsDefaultDomainA->assertZ(newtonsDomain),
  unitsDimensionA->assertZ(Force),
  unitsDefaultAttrA->assertZ(force),
  compositeUnitsBasicUnitToPowerMapA->
   assertZ(<<<* kilograms(1), meters(1), seconds(-2) *>>>),
  nicknameListA->assertZ([ "N"] )
*};
```

The following is a table of the `Dimension` instances that StructProc9 knows by default, and the attributes, default domains, and units of those default domains.

Dimension	Attribute	Default Domain	Units of Default Domain
`dimensionless`			`unitless`
`arc`	`angle`		`radians`
`Distance`	`length`		`meters` (abbr. "`m`")
`Time`	`duration`		`seconds` (abbrs. "`s`", "`sec`")
`Mass`	`mass`		`kilograms` (abbr. "`kg`")
`ElectricalCharge`	`electricalCharge`		`Coulombs` (abbr. "`C`")
`Temperature`	`temperature`		`Kelvins` (abbr. "`K`")
`CountDimension`			`things`
`Moles`	`moles`		`mol`
`Area`	`area`		`squareMeters`
`Volume`	`volume`	`cubicMetersDomain`	`cubicMeters`
`Velocity`	`velocity`		`metersPerSecond`
`Acceleration`	`acceleration`		`metersPerSecondSquared`
`Momentum`	`momentum`		
`Force`	`force`		`Newtons` (abbr. "`N`")
`Energy`	`energy`		`Joules` (abbr. "`J`")
`Power`	`power`		`Watts` (abbr. "`W`")
`Pressure`	`pressure`		`Pascals` (abbr. "`Pa`")
`ElectricalCurrent`	`electricalCurrent`		`Amperes` (abbr. "`A`")
`Voltage`	`voltage`	`voltsDomain`	`Volts` (abbr. "`V`")

To print the justifications associated with annotated values, execute the following.

```
SP9a1 :) [ 1]  thisSE->assert(spEnvsShouldPrintJustificationA,true);
```

```
thisSE
SP9a1 :) [ 2]  4(* *);
4(*limitlessDimensionlessDomain*) <~
^BySaySoOf[*`kbRun_developer_2020-05-28_21:38:46_CDT`*]
```

The printing of justifications can be turned of by typing:

```
SP9a1 :) [ 3]  thisSE->assert(spEnvsShouldPrintJustificationA,false);
thisSE
SP9a1 :) [ 4]  4(* *);
4(*limitlessDimensionlessDomain*)
```

The raise-to-a-power operator is ^ used as a binary operator. The exponent operand (the right-hand-side operand) must be dimensionless.

```
SP9a1 :) [ 47]  2.0(* meters *) ^ 3;
8.(*cubicMetersDomain*)
SP9a1 :) [ 48]  8.(*cubicMetersDomain*) ^ (1\3);
2.(*defaultMetersDomain*)
```

10. Introduction to Arguments, Justifications and Attacks

10.1 Introduction

Key to science is not just an "answer" to a scientific question, but also the reasoning behind it. Also key is the reasoning behind why some other answer should *not* be believed.

SP9A uses *justifications* to support answers and *attacks* to critique them. Both justifications and attacks are subdivided into *instances* and *patterns*. Justification and attack instances are directly associated with answers and do the immediate supporting (or attacking). Patterns, however, are associated with assertions that create answers. Patterns are used to create justification and attack instances to associate with answers once that assertion generates an answer.

For example, consider an assertion representing a least-squares fit of y as a function of x.

X	Y
1.0	9.9
2.0	18.9
5.0	50.4
7.5	74.0
10.0	101.1

Let the assertion state:

R-squared	0.9995
Slope	10.12
Intercept	-0.7544

Although this assertion may have justification and attack instances directly attached to it concerning the validity of the data X and Y, and with the validity of the means by which the least-squares was computed, let us first deal with two attack patterns that could be associated with the least-squares fit. Two attacking patterns could be:

(1) The independent variable had a range from 1.0 to 10.0. Thus, any attempt to use this linear fit for values far outside this range is not supported by this limited range.

(2) The R-squared error is 0.9995.

When we use this assertion to compute Y at X = 9.0 we get 90.3. To this answer we apply both attack patterns and generate two attack instances: (1) *Hey! The source data for X had a limited range from only 1.0 to 10.0!* (2) *Hey! R-squared error is 0.9995!*

In this particular case both (1) and (2) may be considered irrelevant. Our X = 9.0 is in [1.0, 10.0], and an R-squared error of 0.9995 is considered very close to linear. Therefore, in this particular case, our *the attacking instances generated by both (1) and (2) would themselves be attacked by two additional attacking instances!* The purpose of keeping all these attacks is to show "*Yes, we are aware of problems with least-squared fits in general. For this particular case, however, we have shown that they are not applicable.*"

10.2 Ontology

The class `Argument` covers classes `AbstractJustification` and `AbstractAttack`. `AbstractJustification` is further subdivided into `JustificationPattern` and `Justification` (justification instances). Likewise, `AbstractAttack` is further subdivided into `AttackPattern` and `Attack` (attack instances).

10.3 JustificationPattern

`JustificationPattern` covers

10.4 Justification

`Justification` covers justification instances and is not subclassed. The attributes listed for the constructor are:

Attribute name:	Description:
`justificationsSubjectA`	Maps to what has the value that is being justified.
`justificationsAttrA`	Maps to the attribute of the subject being justified.
`justificationsImmediateTypeA`	Tells classification of operator in by justification type. Types include byArithmetic, byConvention, byHypothesis, byAbduction, and byObservation.
`justificationsDependentTypesDsA`	A data-structure that includes that value of justificationsImmediateTypeA and all other type values through-out the tree defined by the operand map.
`operatorA`	Tells computation that was just done to compute justified value or assertion.
`justificationsOperandMapA`	Tells arguments to computation that was just done to compute justified value or assertion.
`justificationsValueA`	Maps to the annotated value or assertion being justified.
`JustificationsCompareOpA`	One of {=,!=, <, <=, >, >=, AND}. The first six of these tell the relationship between (justificationSubjectA,justificationAttrA) and justificationsValueA. For example, “>” means (justificationSubjectA,justificationAttrA) > justificationsValueA. AND signifies that two justifications concerning the same (justificationSubjectA,justificationAttrA) pair are being combined into one larger, conjunction. It is meant to define dual-relations like: value[lo] <= (subject,attr) <= value[hi] out of: value[lo] <= (subject,attr) AND (subject,attr) <= value[hi]

10.5 AttackPattern

Instances of `AttackPattern` are typed by the nature of the attack:

AppliedValueOutsideInducedAssertionRa	

nge	
AssertionInaccurate	The assertion used is inaccurate, e.g. has error bars that are too great.
UnreliableAgent	An agent (“agent” in the A.I. sense: human, intelligent machine, extra-terrestrial intelligence, precocious bonobo, *etc*.) used to generate the value is unreliable.
UnreliableInstrument	
UnreliableComputationalTechnique	
UnwarrantedBecauseOf	An attack pattern that attacks attack instances by claiming the original attack is unwarranted. Refers both to the original attack, and information telling why it is not applicable.

10.6 Attack

`Attack` covers attack instances.

`attackTypeA`	Maps to one of the `AttackPattern` instances
`attackTargetA`	Maps to the assertion being attacked
`attackArgumentsMapA`	Maps to a map that maps supplemental arguments to their values. For example, for `AttackPattern UnreliableAgent`, the map maps `agentA` to the identity of allegedly unreliable agent.

11. A Calculus of Justifications

Justification instances may just be called "justifications" when the reference is not ambiguous with justification patterns. Justifications have the syntax:

```
(annotated value) <~~ (justification instance)
```

Mathematics already has the notation:

```
y <- x
```

to signify that `y` is true when `x` is, and already uses ~ to mean "is similar"[7]. A more precise semantics for "`y <~~ x`" is "*Although x may or may not be able to derive y as the annotated value for the subject and attribute of x, x at least supports the credibility of y being that annotated value*".

In general, a justification instance is a tree of justifying nodes. Each node has its own attributes, which in terms of the tree should not be considered leaves, but just structure within the node itself. For example, a node may group together the subject (*Joe Phillips*), attribute (*mass*) and annotated value (*80.51 kg*). However, when a justifying node has another justifying node as one of its values, then it becomes an internal node of some overall justification tree.

The *intuition* behind math on justified annotated values is that one should get an annotated value that results form the specified math operation, but whose justification combines the justifications of the operands. For example, given an annotation for *80.51 kilograms*:

```
80.51(*kgDomain*)
```

a justification instance for it:

```
joeTellsJoesMass2020Jan22[*ByMeasurement|`Joseph Phillips`,
mass,`Joseph Phillips`,^Date[*2020,1,22*],`Joe's master bathrm`,
`Conair Corp Model WW404GD scale`*];[8]

80.51(*kgDomain*) <~~ joeTellsJoesMass2020Jan22;
```

then multiplying it with an annotated value (`9.81(*metersPerSecSqrDomain*)`) gives another justified annotated value:

```
80.51(*kgDomain*) <~~ joesTellsJoesMass2020Jan22 *
9.81(*metersPerSecSqrDomain*);

789.(*newtonsDomain*) <~~ ^ByMath_1234;
```

The dimensionality (`newtonsDomain`) and value (`789.`) of the resulting annotated value have been calculated to the available precision. Additionally, a new justification instance has been generated:

---

7 Mathematics has `A |= B` that means "*in every model in which A is true, B is also true*", and `x |- y` which means "*y is derivable from x*". However, I wanted a notation where the annotated value being supported comes first.

8 The expression defines `joeTellsJoesMass2020Jan22` as an instance of `ByMeasurement` with subject `` `Joseph Phillips` `` and attribute `mass`. It was made by `` `Joseph Phillips` `` on 2020 January 22, at Joe's master bathroom, using a Conair Corp Model WW404GD scale.

`^ByMath_1234`. The exact name has been system-generated and is not important. What is important, however, is its definition. `^ByMath_1234` tells that it derives from multiplying `80.51(*kgDomain*) <~~ joesTellsJoesMass2020Jan22` by `9.81(*metersPerSecSqrDomain*)`.

One may also combine justified annotated values with other justified annotated values:

```
80.51(*kgDomain*) <~~ joesTellsJoesMass2020Jan22 *
9.80665(*metersPerSecSqrDomain*) <~~ cgpm3DefinesStdG[9];

789.5(*newtonsDomain*) <~~ ^ByMath_1238;
```

The result is similar, another justified annotated value to the given precision. However, this time `^ByMath_1238` knows that it depends on two prior justification instances: `joesTellsJoesMass2020Jan22` and `cgpm3DefinesStdG`.

A justified annotated value (*e.g.* `789.5(*newtonsDomain*) <~~ ^ByMath_1238`) can be thought of as a number (`789.5`) but with units, a dimension (`newtonsDomain`), an optional error range or standard deviation, and a justification instance (`^ByMath_1238`) attached. As such, it can be added, subtracted, multiplied, divided, *etc*.

Syntactically, `<~~` is another annotation to the value, much like "kg" helps describe "80.51". Much like "kg" in "80.51 kg" the justification portion binds closer than any mathematical operators because it truly is part of the value. Thus, this is both syntactically wrong and sematically meaningless:

```
sin(3.141592(*radiansDomain*)) <~~ ^ByWhatever;
```

just as "*sine(3.141592) radians*" makes little sense. The proper form should be:

```
sin(3.141592(*radiansDomain*) <~~ ^ByWhatever);
```

because `sin()` is applied to the whole value: both the units of radians and the justification.

Semantically, while `789.5(*newtonsDomain*)` is an annotated value, `789.5(*newtonsDomain*) <~~ ^ByMath_1238` is a justification with `789.5(*newtonsDomain*)` as its annotated value. Thus, while `789.5(*newtonsDomain*) <~~ ^ByMath_1238` is called a "*justified annotated value*", it is classified as a justification.

What are the rules for combining annotated values with justified annotated values, and for combining justified annotated values with themselves? For ***computing the value and error bars or standard deviation*** the rules are as established by the existing rules of mathematics and precision. For ***computing the domain*** the rules are as established by existing rules of dimensional analysis and unit conversion.

What the remainder of this chapter aims to establish are the rules for ***computing the justification***.

We start with addition, and following Phillips [P10] we note that semantically "addition" can mean at

9 The 3rd Conférence Générale des Poids et Mesures (CGPM) met in 1901 and established the standard that g is defined to be 9.806 65 m/s^2.

least one of two things. With ***grouping addition*** one conceptually creates a bag of the subjects whose attributes are being added, and the sum describes pertains to the bag. For example, consider adding the masses of an adult male rubythroated hummingbird with that of an adult male rufous hummingbird: the sum pertains to the bag of those particular birds.

In contrast with grouping addition is ***extension addition*** where the resulting subject is taken as a modified form of one of the subjects of the addends. For example, the mass of Joe Phillips increases upon eating a rich piece of chocolate cake. The Joe Phillips that just ate the cake is taken as pretty much the same of entity as the Joe Phillips from before the cake.

Thus, we need two different addition operators to address these two different cases. I propose that the plus sign (+) denote extension addition, and that the n-ary function `sum()` denote grouping addition. While the new subject of grouping addition is the bag of all subjects, the subject of extension addition is the subject on the left hand side.

Please note, both `2.0(*kgDomain*) + 2.0(*kgDomain*)` and `sum(2.0(*kgDomain*),2.0(*kgDomain*))` compute the same annotated value (`4.0(*kgDomain*)`). They only differ in how they handle the subject meta-data of the justification.

```
sum(3.0(*gramsDomain*) <~~ ^ReportedAdultRubythroatMass,
    3.9(*kgDomain*) <~~ ^ReportedAdultRufousMass) =
3.9(*kgDomain*) <~~ ^BagOfReportedMasses

80.51(*kgDomain*) <~~ ^ReportedJoePhillipsMass +
 120(*gramsDomain*) <~~ ^ReportedCakeMass  =
80.63(*kgDomain*) <~~ ^NewComputedJoePhillipsMass
```

Grouping addition

The sum resulting from a grouping addition is a property of the group, not its individuals. The function `group()` returns an entity that represents the group as a whole:

```
group({ rubythroatedHummingbird67,rufousHummingbird77} )
```

As a bonus we can use the same function for attributes. For example, consider the attribute that results from adding the *height* of Joe Phillips with the *width* of Joe Phillips:

```
group({ height,width} )
```

Extension addition

Like grouping addition, extension addition uses the function `group()` to compute the resulting attribute. However, extension addition ***uses the subject of the left-hand-side addend as the subject of the sum:***

```
x <~~ JusticationWithSubjectA + y <~~ JusticationWithSubjectB =
(x+y) <~~ JusticationWithSubjectA
```

The reason for this is to agree with formulas of the general form:

$$x_{new} = x_{old} + \Delta x$$

because, generally, x_{new} is taken to be very similar to x_{old}, but with some small, defined change.

Note, for the *subject*, (+) addition is still associative:

```
(x <~~ JustSubjA + y <~~ JustSubjB) + z <~~ JustSubjC
= x <~~ JustSubjA + (y <~~ JustSubjB + z <~~ JustSubjC)

(x+y)<~~JustSubjA + z<~~JustSubjC =
 x<~~JustSubjA + (y+z)<~~JustSubjB

(x+y+z) <~~ JustSubjA = (x+y+z) <~~ JustSubjA
```

However, it is no longer commutative:

$$x\text{<~~JustSubjA} + y\text{<~~JustSubjB} \neq_{subj} y\text{<~~JustSubjB} + x\text{<~~JustSubjA}$$

$$(x+y) \text{ <~~JustSubjA} \neq_{subj} (y+x) \text{ <~~ JustSubjB}$$

Here $\neq_{subj}$ denotes the inequality of the *subjects* on both sides. Arithmetically, `(x+y)` of course, still equals `(y+x)`.

Unfortunately, this rule could disagree with the intuition behind common polynomial notation. Polynomials are generally written in order of decreasing powers.

$$Ax^2 + Bx^1 + Cx^0 = Ax^2 + Bx + C$$

Here, the constant `C` is the fixed thing that establishes the value to which the squared and linear terms are added. Thus, it makes sense to have `C` have the subject. Unfortunately the subject of the resulting sum will come from `A`.

However, it *agrees* with when polynomials are written in the reverse order. An important example of this is the Taylor series used for approximating a smooth function around a point when the value of that function at the point, and some of its derivatives are known:

$$f(x) = f(x_0) + f'(x_0)*(x-x_0) + f''(x_0)*(x-x_0)^2/2 + f'''(x_0)*(x-x_0)^3/6 + \ldots$$

Again, the subject of `f(x)` is taken to be that of $f(x_0)$, with some potentially minor changes made.

Grouping and extending subtraction

In analogous form to grouping addition using `sum()` and extending addition using (+), grouping subtraction uses `diff()` and extending subtraction uses (−). The function `diff(x,y)` computes the quantity (x-y) and is equivalent to `sum(x,-y)`. The operation `(x-y)` computes the quantity (x-y) and is equivalent to `(x + -y)`.

Grouping multiplication

Grouping addition uses `sum()` and grouping multiplication uses `product()`. The subject of the justification is handled exactly the same way as with `sum()`: by creating a bag of the multiplied subjects. The attribute of the justification, however, changes with dimensionality and units according to what has been multiplied.

As an example, say one wants to compute the probability of obtaining a “6” on the roll of a single fair die, ***and*** obtaining a “heads” of the flip of a fair coin, ***and*** obtaining a “6 of clubs” when choosing a card at random from a standard set of playing cards:

```
prod(1\6 <~~ ^ByMath(^FairChoice(die,6)),
     1\2 <~~ ^ByMath(^FairChoice(coin,heads)),
     1\52 <~~ ^ByMath(^FairChoice(deckOfCards,clubs6)) =
1\624 <~~ ^ByMath({ ^FairChoice(die,6),
 ^FairChoice(coin,heads),^FairChoice(deckOfCards,clubs6)})
```

The resulting justification has references the bag `{ ^FairChoice(die,6), ^FairChoice(coin,heads),^FairChoice(deckOfCards,clubs6)}`. Using an non-ordered bag is appropriate because the ordering of the die-rolling, coin-flipping and card-picking does not matter.

Scaling multiplication

Scaling multiplication (*) is similar to extension addition (+) in that the subject in the justification of the output comes from just one operand. However, for scaling multiplication the subject of the product becomes the subject of the ***right-hand-side*** argument, while the extension addition the subject of the sum was the subject of the left-hand-side.

The reason for using the right-hand-side is that simple multiplications by a constant are often written as:

```
product = constant * factor
```

where `factor` contains the specific details of the thing whose quantity is being multiplied. By getting the right-hand-side subject, we pass the subject along to the product.

Common equations where this policy works well include:

Hooke's Law:	`f(x) = -kx`
Diameter of a circle:	`d = 2πr`

There are, however, cases that require some re-thinking, if not re-writing. Consider weight = mass * (gravitational acceleration):

```
w = mg
```

Our policy would obtain the subject of `w` from that of `g`. If g is thought of as pertaining to the Earth, that would be bad because it would appear to attempt to compute the "weight of the Earth on Earth". However, if instead we think of `g` as being "*the acceleration due to gravity of the mass whose subject is m*", then we compute the correct subject.

This being careful of exactly what `g` means helps us in other ways too. The experienced value of `g` differs slightly according to altitude and position on Earth. By making `g` position-dependent, we also (potentially) make it more accurate.

Now consider the Ideal Gas Law:

```
PV = nRT
```

On the left-hand-side we have `pressure*volume`. If we naively think of this as the "*volume of the container*" then we have computed a property (thermal energy) of the container, and not the contained gas. Instead, we should think of `v` as being "*the volume of the gas held in the container*" (which in equilibrium is just the volume of the container itself). Then we compute the thermal energy of the gas.

Now consider the right-hand-side: `(number of moles of gas)*(gas`

`constant)*(absolute temperature of gas)`. The gas constant is a constant, so a better way to write it would be:

```
PV = RnT
```

or:

```
PV = RTn
```

Unfortunately, there are cases were the conventional form should be re-written. Consider:

$d_{new} = d_{old} + r * \Delta t$

Extension addition properly gives d_{new} the same subject as d_{old}. The multiplication, however, gets its subject from the right-hand-side of the product: Δt. This is just some time step assumed by the simulation or process; it is not inherent to the thing whose motion is being calculated. But `r` is. Thus, the proper way to write this formula is as:

$d_{new} = d_{old} + \Delta t * r$

Grouping and scaling division

In analogous form to grouping multiplication using `product()` and scaling multiplication using (`*`), grouping division uses `div()` and scaling division uses (`/`). The function `div(x,y)` computes the quantity (x/y) and is equivalent to `product(x,`y^{-1}`)`. The operation `(x/y)` computes the quantity (x/y) and is equivalent to `(x*`y^{-1}`)`.

Much like for grouping and scaling multiplication, for grouping and scaling division the attribute that results depends upon what was divided.

12. Scientific Knowledge Representation

12.1. Introduction

Finally, we have enough background to state knowledge beyond just specific and inherited values! Let us start with the Ideal Gas Law:

$$PV = nRT$$

Note that "R" is a constant. Therefore, as discussed in Chapter 11, a better way to represent it given the right-hand-side attribute passing nature of scaling multiplication is as:

$$PV = RnT$$

Where should we place this equation? The equation is ***for*** gases, but it ***is*** an equation of state. Thus, the recommended Scienceomatic policy is to define it in a class `BeingAGas`, which is subclass of `Event`. The class `Event` covers processes and states, and defining such equations under it consolidates such knowledge.

12.2. Entities and Values

The equation implies the existence of both a gas and a container for it. The equation interrelates the pressure, volume, quantity and temperature of a gas in equilibrium. Further, to be in equilibrium with some fixed volume implies a container of fixed internal volume (which the gas completely fills, because it is a gas).

We introduce the gas and its container with these statements given within the explicit construction of `BeingAGas`:

```
(?this,gas)->subConstEntity(?gas,Gas);
(?this,container)->subConstEntity(?container,PhysicalObject);
```

The morphology `?var` signifies a ***structural variable***. Structural variables represent values mentioned by the structure. Here, `?this` represents some particular instance of being a gas, `?gas` represents the gas and `?container` represents the container.

In the sub-expression `(?this,gas)`, `gas` is an attribute of the structure (in this case, `BeingAGas`). The full expression:

```
(?this,gas)->subConstEntity(?gas,Gas);
```

binds the new variable `?gas` as a stand-in for whatever value denoted by attribute `gas` of `?this`. Further, this value is limited to be within class `Gas`. The "sub" of `subConstEntity()` means that `?gas` stands in for a gas at instances of this class, not this class itself. The "Const" means that the gas does not change. The "Entity" means that `?gas` represents an empirical entity or cultural convention (*i.e.* `Gas` must be a subclass of either `EmpiricalEntity` or `CulturalConvention`). The following methods exist to declare structural variables:

Method	Purpose: To declare a structural variable that ...
`constEntity`	Is constant and ranges over `EmpiricalEntity` or

Method	Purpose: To declare a structural variable that ...
subConstEntity	`CulturalConvention` for `?this`.
	Is constant and ranges over `EmpiricalEntity` or `CulturalConvention` instances for `?this`.
constValue	Is constant and ranges over `Dimension` for `?this`.
subConstValue	Is constant and ranges over `Dimension` instances for `?this`.
varValue	Is variable and ranges over `Dimension` for `?this`.
subVarValue	Is variable and ranges over `Dimension` instances for `?this`.

In principle, there could also be `varEntity` and `subVarEntity` to represent variables that range instances of `EmpiricalEntity` and `CulturalConvention`. This however, implies that a subject itself is changing in some fundamental fashion. For now, subjects must be constant.

Now that we can refer to the gas instance as `?gas`, we can introduce the four variables of the equation:

```
(?gas,pressure)->subConstValue(?pressure,Pressure);
(?gas,volume)->subConstValue(?volume,Volume);
(?gas,numMoles)->subConstValue(?numMoles,NumMoles);
(?gas,temperature)->subConstValue(?temp,Temperature);
```

The first line declares that `?gas` should have a constant attribute (`pressure`), that will be denoted with `?pressure`. Further, this value is in dimension `Pressure`, which is a numeric value (as denoted by "Value" as opposed to "Entity"). The other three are similar.

Note that `?volume` refers to the volume of the gas, not that of the container. Let us introduce another variable to represent the internal volume of `?container`.

```
(?container,internalVolume)->subConstValue(?innerVolume,Volume);
```

We will tie these two volumes together in an equation shortly.

12.3. Relations

We may now state the Ideal Gas Law equation:

```
relationList->
  subWacati
        (idealGasLawEqn
         [*Relation |
           ?pressure * ?volume = gasConst * ?numMoles * ?temp,
           [], // Justification list
           []  // Attack list
         *]
        );
```

The `subWacati()` statement creates a vector list for the inherited `relationList` if it does not have a value. Then it inserts its argument (in this case the `Relation` instance) into this list. If `relationList` already has a value but it is not a data-structure upon which `insert()` may be

done, then an exception is thrown. The term *wacati* is short for "*when absent create; and then insert*"[10]. Besides assertion methods `wacati()` and `subWacati()`, there are also `wacatiA()` and `subWacatiA()`. They insert at the beginning of the lists, just like `insertA()`.

The argument to this `subWacati()` is a relation. There is also just `wacati()`, which also may take a relation as an argument. Relations relate two expressions of defined structural variables, constants, and mathematical operators. The highest-level relation operator must be one of the following: =, << (is much lesser than), <, <=, >, >> (is much greater than) or >=.

The constant `gasConst` is actually a named justification for why the gas constant has the value that it does.

The constructor for `Relation` lists these attributes:

`assertionsExpressionA`: maps from the relation instance to the expression that represents it.

`assertionsJustificationListA`: maps from the relation instance to a list of justifications for that particular relation (as opposed to the larger structure that it is in). Such justifications could include the derivation of the expression.

`assertionsAttackListA`: maps from the relation instance to a list of attacks for that particular relation (as opposed to the larger structure that it is in). Such attacks could include the assumptions that had to be made to derive the expression.

12.4. Implication Sentences and Conditions

Events often have preconditions and assumptions associated with them. Conditions exist to state such knowledge. And one of the most common form of condition is the implication sentence.

Implication sentences have general form:

```
forAll(?var0,Class0),
forAll(?var1,Class1):
...
  pred0(...),
  pred1(...)
  ...
  =>
  consequent(...)
```

There may be zero or more `forAll` clauses before the colon, each of which introduces a variable that is restricted to the specified class. (If there are no `forAll` clauses then the colon is absent.) The predicate clauses before the implication tell a conjunction of conditions. The philosophy of these clauses is that they either introduce a variable over a broad class, or they restrict a previously introduced variable to the domain of interest. Actually assigning of values ought to be done by Skolem constants and functions in the implied clauses. Supported predicates include:

`isMemberOf(instance,composition)`: That `instance` is a member of the `composition`. For example, a fuselage is a member of the composition for an airplane.

`isInstanceOf(instance,class)`: That `instance` is an instance of `class`.

`isSubclassOf(subclass,superclass)`: That `subclass` is a subclass of `superclass`.

`inTimeRange(subRange,superRange)`: That `subRange` is a time range wholly within `superRange`.

10 It follows in the tradition established by the computing term WYSIWYG being short for "*what you see is what you get*". And, "wacati" is another verb!

`event(name, referenceFrame, timeFrame, entityList, propertyList)`: That the `Event` instance named `name` with reference frame `referenceFrame` and time frame `timeFrame` exists. It interrelates the entities of `entityList` given their properties as given in `propertyList`. The members of `entityList` may either be `entity(?variable,Class)`, which introduces a new variable `?variable` to be restricted to membership in class `Class`, or `distinct(?variable,Class)` which does the same, with the caveat that whatever `?variable` stands for is distinct from any other stood-for thing in the entity list.

Supported action predicates include:

`prop(subj,attr,val)`: That subject `subj` has attribute `attr` value `val`. If `subj` and `attr` are either constants or bound variables then `val` may be new variable; the clause binds that variable to the computed value.

Supported Skolem functions include:

`val(subj,attr)`: Returns (or syntactically behaves as if it returns, à la Prolog) the value that subject `subj` has for attribute `attr`.

With that established we may now approximate some assumptions of the Ideal Gas Law equation:

(1) "The gas consists of molecules whose collective volume is much smaller than the volume of the container"
(2) "The molecules have the same mass"
(3) "The molecules are spherical and collide elastically with themselves and the walls of the container"
(4) "The molecules do not interact with each other except by elastic collisions. No inter-molecular forces, relativistic effects, quantum effects. The equations of motion are reversible."
(5) "Average kinetic energy of molecules depends only on the absolute temperature of the system."

"*The gas consists of molecules whose collective volume is much smaller than the volume of the container*" can be represented with:

```
(?gas,molecularComposition)->subConstEntity(?gasMolecule,Molecule);
(?gasMolecule,volume)->subConstValue(?molecularVolume,Volume);
conditionList->
  subWacati
        (^Relation
        [* avogadrosConst * ?numMoles * ?molecularVolume
           << ?innerVolume
        *]
        );
```

The first two `subConstDecl()` statement introduce `?molecularVolume`. We need that to define the << inequality relation that is the argument to `subWacati()` statement.

"*Molecules have the same mass*" can be approximated with "*For all molecules, if they are members of the molecular decomposition of the gas then that molecule has the mass as the first returned member of the that molecular decomposition.*"

```
conditionList->
```

```
    subWacati
    (forAll(?mol,Molecule):
     isMemberOf(?mol,val(?gas,molecularComposition))
     =>
     prop(?mol,
          mass,
          val(val(val(?gas,molecularComposition),memberOf),mass)
         )
    );
```

"*Molecules are spherical ...*" can be represented with "*For all molecules, if they are members of the molecular decomposition of the gas then that molecule has the shape of a sphere.*":

```
conditionList->
    subWacati
       (forAll(?mol,Molecule):
        isMemberOf(?mol,val(?gas,molecularComposition)
        =>
        prop(?mol,shape,sphere)
       );
```

"… and collide elastically with themselves and the walls of the container. Molecules do not interact with each other except by elastic collisions. No inter-molecular forces, relativistic effects, quantum effects. The equations of motion are reversible." can be approximated with:

```
conditionList->subWacati
    (forAll(?event,Event):
       event(?event,                              // Name
             ??,                                  // Reference frame
             ?eventTime,                          // Time frame
             [ distinct(?molecule0,??),           // Entity list
               distinct(?molecule1,??)
             ],
             ??                                   // Property list
            ),
       inTimeRange(?eventTime,?time),
       instanceOf(?molecule0,Molecule),
       instanceOf(?molecule1,Molecule),
       isMemberOf(?molecule0,val(?gas,molecularComposition)),
       isMemberOf(?molecule1,val(?gas,molecularComposition))
       =>
       instanceOf(?event,ElasticCollision)
    );
```

The implication sentence says "*For all events that mention only two distinct molecules in the gas that happen during the time of the outer* `BeingAGas` *event, those events are instances of* `ElasticCollision`". The sentence uses variables `?time` and `?gas` that are bound outside the scope of the sentence, and `?event`, `?molecule0` and `?molecule1` that are bound inside. The variable `??` is like the variable "_" in Prolog: it matches anything, and different occurrences in the same sentence are free to match different things. There is a similar sentence for events between gas molecules and the container.

"*Average kinetic energy of molecules depends only on the absolute temperature of the system*." can be written as:

```
conditionList->
  subWacati(^Relation
            [* sumOver
                (?molecule,
                 ?gasDecomposition,
                 ?molecularMass * val(?molecule,velocity)^2 / 2
                )
                / avogadrosConst * ?numMoles
                = FO(?temp)
             *]
            );
```

Inside of the summation-computing `sumOver(?var,Class,expression)` clause `val(subj,attr)` is being used as a two-argument function that retrieves values. There is also the analogous product-computing clause `prodOver(?var,Class,expression).` Both `sumOver()` and `prodOver()` only bind their variables in that clause itself.

The expression `FO()` will be explained in the next sub-chapter.

12.5. FO() and fo()

The purpose of `FO()` and `fo()` is to restrict the future algebraic form of some expression. They are named "fo" for "function-of" and have a syntax motivated by O-notation in computer science. For example, to say function f(n) *could* be written as $An^2+Bn+C+g(n)$, one may write:

$f(n) = fo(n^2+n+1)$

While to say f(n) *must* be written as An^2+Bn+C, one writes:

$f(n) = FO(n^2+n+1)$

f(n) = FO(1) means "f(n) is constant", while f(n) = fo(1) tells you nothing about f(n).

12.6. Reasoning

Computation is done using annotated-values (non-optimized) or non-annotated (optimized) when walking an expression tree in a relation. The justification is created and attached to the final computed annotated value. This justification is then attacked by realized conditions from both the relations that were used to compute the annotated value and the conditions of the `Event` instances.

For example, if we are given that a particular simple of gas has:
volume = 2.00 liters
temperature = 300 Kelvin
quantity = 0.200 moles
then the computed pressure would be 9.98e+5 Pascals (`9.98e+5(*Pascals*)`). The resulting justification would be attacked by the six[11] conditions mentioned section 12.4, with external variables substituted with the particular values for the gas sample.

11 Five listed one, plus an `ElasticCollision` one for gas molecule/container events similar to the given gas molecule/gas molecule events.

The system call `support(attackedJustification,maxNumAttacks)` tries to create a new justification equivalent to `attackedJustification` but with no more than `maxNumAttacks` attack instances against it. It uses an algorithm like the one listed below:

```
Justification support
        (Context:            context,
         Justification:      attackedJustification,
         NonNegativeInteger: maxNumAttacks
        )
{
  numAttacks   = attackedJustification.attackList.length;
  maxDepth     = context.maxSearchDepth;
  currentDepth = 1;

  while  (numAttacks <= maxNumAttacks)
  {
    if  (currentDepth > maxDepth)
      return null;

    for  (attack in attackedJustification.attackList)
      if  ( canDefendAgainst(attack.statement,currentDepth) )
         attackedJustification.attackList.remove(attack);

    currentDepth++;
    numAttacks   = attackedJustification.attackList.length;
  }

  return attackedJustification;
}
```

Here, `canDefendAgainst()` tries to prove that the necessary expression holds using the given maximum search depth.

The algorithm listed above is a simplification in that once attacks have been successfully counter-attacks, the attacks do not go away. Instead, the counter-attack is attached to the attack, and the original attack is no longer considered a threat.

Reasoning uses knowledge from specific to general. Therefore:
(1) The context queried first
(2) Knowledge given with a subject is queried second
(3) Inherited knowledge given at classes of the subject are queried from the most specific class up to `Idea`.
(4) At `Idea`, analytical knowledge, *e.g.* knowledge of how to recast problem (definitions of predicates, switching between Cartesian and Polar coordinates) should be stated

StructProc9 borrows both syntax and semantics from Prolog by using Skolem functions. In Prolog, Skolem functions are a syntactic trick. The Prolog reasoning engine manipulates expressions so that functions are never “called” in the sense of being executed as in imperative languages. StructProc9, however, can either manipulate logic sentences syntactically (as in Prolog), or call them as true functions.

12.7. Parameterized Equations

Now we consider parameterized equations, like acceleration, velocity and position as a function of time. First, let us introduce class `Event`.

```
class  Event
{*
  isA->assertZ(EmpiricalEntity),

  (?this,refFrame)->subConstEntity(?refFrame,ReferenceFrame),

  (?this,when)->subConstValue(?tcc,Time),
    //  ?tcc = calendar/clock time:
    //          when Event started according to a global chronology

  (?this,time)->subVarValue(?tsw,Time)
    //  ?tsw = stop-watch time: time during event
*};
```

Event introduces three structural variables for all classes that derive from it: `?refFrame`, `?tcc` and `?tsw`. `?refFrame` tells the reference frame. The reference frame is a fixed cultural convention that is held by instances of `Event`, so we introduce `?refFrame` with `subConstEntity()`. `?tcc` stands for "calendar/clock time" and tells the fixed time that the event instance started according to some external notion of time. `?tsw` stands for "stop-watch time" and represents the flow of time during the event. This starting time is a fixed value, so we introduce the variable with `subConstValue()`. `?tsw` is not fixed (except, perhaps for instantaneous events), so we introduce it with `subVarValue()`.

Now that we have introduced Event, we can an introduce immediate subclass of it.

```
class OneDimensionalMotion
{*
  isA->assertZ(Event),

  (?this,movedThing)->subConstEntity(?movedThing,PhysicalObject),
  (?movedThing,acceleration)->subVarValue(?a,Acceleration,[?tsw]),
    //  ?a is in dimension Acceleration, and varies as a fnc of ?tsw
  (?movedThing,velocity)->subVarValue(?v,Velocity,[?tsw]),
  (?movedThing,position)->subVarValue(?p,LinearPosition,[?tsw]),

  relationList->
    subWacati(^Relation[* ?v[?tsw] = integral(?a,?tsw) *]),
    // This equation is not yet usable because we have not been
    // given ?a as a function of ?tsw
  relationList->
    subWacati(^Relation[* ?p[?tsw] = integral(?v,?tsw) *])
*};
```

Class `OneDimensionalMotion` talks about the motion of something (`?movedThing`) in one dimension. In particular it introduces 3 variables of `?movedThing`: its acceleration (`?a`), velocity (`?v`) and position (`?p`). When we introduced variable ?tsw we did so with `subVarValue(?tsw,Time)`: `?tsw` is an *independent* value in dimension `Time`. Variables `?a`, `?v` and `?p`, however, are *dependent* variables. In particular, they are vary with time `?tsw`. We denote this with the third argument to the `subVarValue()` method call: `subVarValue(?a,Acceleration,[?tsw])`.

Inside of the square brackets is an ordered list of variables that `?a` is taken to be a function of. There is just one: `?tsw`.

Further, we interrelate `?a` with `?v`, and `?v` with `?p` with two relations. In plain mathematical notation those two relations state:

$$v(t) = \int a(t) \cdot dt$$

and

$$p(t) = \int v(t) \cdot dt$$

These formulas are analytically true, and useless without knowledge of a(t) or v(t).

Let us make them useful by defining a simple case: a(t) = constant:

```
class UniformAccelerationOneDimensionalMotion
{*
  isA->assertZ(OneDimensionalMotion),

  relationList->
    subWacati(^Relation[* ?a[?tsw] = FO(1) *])
    //  System has been told ?a is constant, i.e. ?a[?tsw] = ?a[0]
    //    (= ?a, now that it is constant no need to parameterize)
    //  Knowing also that ?v = integral(?a,?tsw),
    //  it is able to compute: ?v[?tsw] = ?a * ?tsw + ?v[0]
    //  Likewise, it also knows:
    //     ?p[?tsw] = 0.5*?a*?tsw**2 + ?v[0]*?tsw + ?p[0]
*};
```

We use the FO(1) notation to fix `?a` to some constant value. This has immediate consequences. The two integrals in the definition of `OneDimensionalMotion` are now analytically computable. Of course, to make specific calculations we must specify `?a`, `?v[0]` and `?p[0]`.

```
dropBall15
{*
  instanceOf->assert(OneDimensionalMotion),

  movedThing->assert(ball15),
  relationList->wacati(^Relation[*?a   = -9.8(* "meters/sec^2" *)*]),
  relationList->wacati(^Relation[*?v[0(*sec*)] = 0(*"meters/sec"*)*]),
  relationList->wacati(^Relation[*?p[0(*sec*)] = 10.0(* meters *)*])
*};
```

With *instance* `dropBall15` we assert that attribute `movedThing` has value `ball15`. This fixes the variable `?movedThing` introduced by `subConstEntity(?movedThing,PhysicalObject)` back in `OneDimensionalMotion`. (We can also verify that `ball15` is an instance of `PhysicalObject`.) ?a, ?v and ?p were all introduced as varying parameterized values, so we fix them with three equations.

We may introduce another subclass where the acceleration is not just constant, but zero. Therefore, the velocity is constant.

```
class ConstantVelocityOneDimensionalMotion
{*
  isA->assertZ(UniformAccelerationOneDimensionalMotion),

  relationList->subWacati(^Relation[* ?a = 0.0(*"m/s^2"*) *])
    //  Told ?a = 0, therefore can compute:
    //    ?v[ ?tsw]  = ?a*?tsw + ?v[ 0]  = ?v[ 0]
    //              (= ?v, no need to index)
    //    ?p[ ?tsw]  = 0.5*?a*?tsw**2 + ?v[ 0]*?tsw + ?p[ 0]
    //               = ?v[ 0]*?tsw + ?p[ 0]
*};
```

All of this is nice, but it is in one dimension. We generalize to two dimensions with the following:

```
class TwoDimensionalMotion
{*
  isA->assertZ(Event),

  (?this,xMotion)->subConstEntity(?xMotion,OneDimensionalMotion),
  (?this,yMotion)->subConstEntity(?yMotion,OneDimensionalMotion),

  (?this,refFrame)->subEquivalent(?xMotion,refFrame),
  (?this,refFrame)->subEquivalent(?xMotion,refFrame),
    //  Ties reference frames of ?this, ?xMotion and ?yMotion
    //    together

  (?this,when)->subEquivalent(?xMotion,when),
  (?this,when)->subEquivalent(?xMotion,when),
    //  Ties starting times of ?this, ?xMotion and ?yMotion
    //    together

  (?this,movedThing)->subEquivalent(?xMotion,movedThing),
  (?this,movedThing)->subEquivalent(?xMotion,movedThing),
    //  Ties moved thing of ?this, ?xMotion and ?yMotion
    //    together

  relationList->subWacati(^Relation[* ?tsw = ?xMotion->?tsw *]),
  relationList->subWacati(^Relation[* ?tsw = ?yMotion->?tsw *]),

  relationList->subWacati(^Relation[* ?ax = ?xMotion->?a *]),
  relationList->subWacati(^Relation[* ?vx = ?xMotion->?v *]),
  relationList->subWacati(^Relation[* ?x  = ?xMotion->?p *]),

  relationList->subWacati(^Relation[* ?ay = ?yMotion->?a *]),
  relationList->subWacati(^Relation[* ?vy = ?yMotion->?v *]),
  relationList->subWacati(^Relation[* ?y  = ?yMotion->?p *])

*};
```

Two dimensional motion is just considered to be two one dimensional motions. (Still have to say that the motions are on orthogonal axes.) The `subEquivalent()` methods assert that the reference frames, starting times and moved-things of both one dimensional motions is that same as that of the two dimensional motion. We also introduce two equations that tie all three `?tsw` times together. Lastly, we introduce 3 new variables so we can talk about the acceleration, velocity and position on the two different axes independently.

Now, let us use this new class as the base class for describing uniform velocity along one axis but uniform acceleration along another. Such a class could describe ballistic motion neglecting wind resistance (*e.g*. throwing a ball on the moon).

```
SimpleBallisticMotion
{*
  isA->assertZ(TwoDimensionalMotion),

  (?this,xMotion)->
 subConstEntity(?xMotion,ConstantVelocityOneDimensionalMotion),
  (?this,yMotion)->
 subConstEntity(?yMotion,UniformAccelerationOneDimensionalMotion)
*};
```

We can also use our two dimensional motion to describe uniform circular motion.

```
UniformCircularMotion
{*
  isA->assertZ(TwoDimensionalMotion),

  (?this,radius)->subConstValue(?r,Length),
  (?this,angularVelocity)->
      subConstValue(?angleVelo,AngularVelocity),
  (?this,angle)->subVarValue(?theta,Angle,[?tsw]),

  relationList->
    subWacati(^Relation[* ?theta[?tsw]
                          = ?angleVelo * ?tsw + ?theta[0(*sec*)]
                        *]
             ),

  relationList->
    subWacati(^Relation[* ?x = ?r * cos(?theta[?tsw])*]),
  relationList->
    subWacati(^Relation[* ?y = ?r * sin(?theta[?tsw])*])
*};
```

For uniform circular motion it is a lot easier to describe the motion in terms of a constant angular velocity, and initial radius and angle. The `?x` and `?y` positions can then be thought of as cosine and sine functions respectively, both as a function of time.

13. Hypothetical Knowledge

13.1 Introduction

The purpose of the knowledge base is to hold knowledge that is now, or once was, believed. Scientists, however, are also interested in "what-if" questions. For example, "*What if the mass of the Sun was a tenth of its true value? How would the Solar System behave?*" We are not (permanently) asserting that the mass of the Sun has changed, we are just asserting a value for the sake of argument.

`AssumeDo` blocks exist to support this reasoning. Like `Do` blocks, the code to run is given in a square-bracket ([..]) delimited `VectorList` instances. Unlike `Do` blocks, however, `AssumeDo` statements have two such lists:

```
^Hypothesize
[*
  [
    // Assumptions are stated here
  ],
  [
    // Calculations are given here
  ]
*];
```

The first list is the "assume" part and is used to state hypothetical values. It may also be used to define new, hypothetical entities (all new hypothetical entities must be subclasses or instances of `EmpiricalEntity`).

The calculations to do with those assumptions are given in the second part, the "do" part. No new entities are allowed to be created in the "do" part. There are two exceptions to this. First, of course, the annotated values, and their associated justifications and attacks that are generated by the calculation are allowed (and will) be calculated. The second is that when `AssumeDo` blocks are nested, the nested ones should be in the "do" portion. Nesting is covered below.

`AssumeDo` is defined not to make any changes outside of itself to the wider knowledge base. The "assume" and "do" portions may ***query*** the larger knowledge-base, but neither may ***change*** it. Entities defined in the "assume" portion are called "hypotheticals".

Because `AssumeDo` blocks do not change the wider knowledge-base, anything computed by them ought to be output by `stdOut->printLn()` or such. Otherwise they may return the Identity value of a hypothetical that will be unknown to the knowledge-base proper, and that will be printed as an unintelligible sequence of hexadecimal numbers.

`AssumeDo` blocks may be nested. When this is done, the inner `AssumeDo` constructs should be given in the "do" portion of the outer ones.

```
^AssumeDo
[*
  [
    // Assumptions A
  ],
  [
    // Calculations are given here
    ^AssumeDo
    [*
```

```
      [
        // Assumptions B1
      ],
      [
        // Calculations assuming A and B1
      ]
    *],
    ^AssumeDo
    [*
      [
        // Assumptions B2
      ],
      [
        // Calculations assuming A and B2
      ]
    *]
  ]
*];
```

Here the “do” portion of the first inner `AssumeDo` block assumes both `A` and `B1`. By contrast, the “do” of the second block assumes both `A` and `B2`. It is legal to nest `AssumeDo` blocks because they are defined not to make any changes outside of themselves. Inner ones have no effect on outer ones.

`AssumeDo` blocks are computationally limited by design. In particular, only the last asserted value will be held for any property. Within the `AssumeDo` block, `assertX()` acts like `assert()` in that it erases previous values. However, taking the `AssumeDo` block and knowledge-base as a whole, `assertX()` acts like `assertA()` in that it does not erase the values found in the knowledge-base, but it does “mask” them. Values asserted in `AssumeDo` are seen before values present in the outer knowledge-base.

14. Justifications

14.1 Introduction

15. Knowledge Base Runs

15.1 Introduction

The work one does may be saved after changes have been made. One may later load this work to continue altering and extending it. However, after one saves one's work as *finished* it becomes immutable as a *finished knowledge base run*. Although finished knowledge base runs may no longer be changed, they have two properties that unfinished ones do not have:

1. They can be locally or globally *published*: made accessible to users of defined groups, or indeed to everyone on the same Scienceomatic environment.
2. They can be *extended*: new knowledge base runs can be built on top of them.

Indeed, a Scienceomatic knowledge base ***is*** a directed acyclic graph of knowledge base runs. In this graph the sole node without any other nodes pointing to it represents the only knowledge base run that is allowed to change (if it has not already been finished). This is the *active knowledge base run*.

Knowledge base runs have both names and numeric indices for the knowledge base runs upon which they are immediately based. Take, for example, the knowledge base comprised of 7 knowledge base runs shown in Figure 13-1.

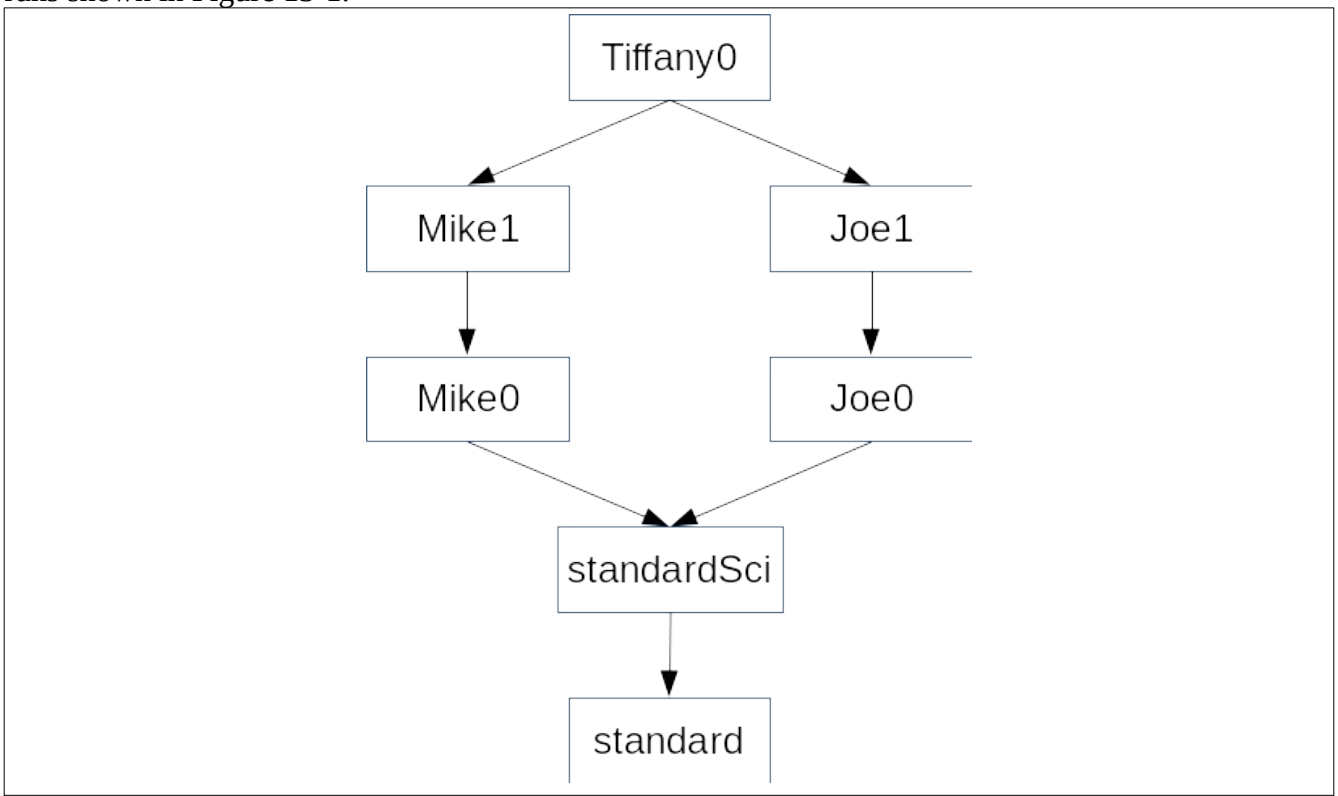

Figure 13-1: A knowledge base of 7 knowledge base runs

Knowledge base runs always reserve index `0` and knowledge base run name `baseEnv` for the sole root of the directed acyclic graph. They always reserve index `1` and knowledge base run name `baseSci` for the sole knowledge base run directly pointing to the root `baseEnv`. Further, the knowledge base run name `thisRun` always refers to the knowledge base run in which it occurs. In the example above, the knowledge base run `standard` has just one entry in its table of knowledge base runs, one for itself at index `0`. As it is its only entry in its table of knowledge base runs, it can

refer to itself by both names baseEnv and thisRun. The knowledge base run standardSci has two entries in its table: baseEnv for the one at index 0, and either thisRun or baseSci for itself at index 1.

Knowledge base runs Mike0 and Joe0 have three entries in their tables: one for baseEnv at 0, one for baseSci at 1, and one for themselves (named thisRun in both) at index 2. Knowledge base runs Mike1 and Joe1 have four entries in their tables: one for baseEnv at 0, one for baseSci at 1, one for the knowledge base upon which they are directly based (either Mike0 and Joe0), at index 2, and one for themselves (named thisRun in both) at index 3. The table for knowledge base run Tiffany0 has 5 entries: baseEnv at 0, baseSci at 1, Mike1 (at either 2 or 3), Joe1 (at either 2 or 3), and one for itself (thisRun) at index 4.

The purpose of knowledge base run baseEnv is to define the environment, including the base ontology and behavior of sp9a. It defines fundamental concepts like Integer, Boolean, String, true, false, *etc*., and serves as the "operating system" of a knowledge base. The purpose of knowledge base run baseSci is to define the basic scientific concepts that subsequent knowledge base runs will extend in the subontology EmpiricalEntity. These include basic scientific concepts like Electron, Protein, Bacteria, Population, Ecosystem, Planet, and Universe.

Different baseSci knowledge base runs imply different scientific worldviews, *e.g*. by what they define (and what they do ***not*** define) in their ontologies. The scientific worldviews of Ptolemy, Newton, Einstein and Heisenberg appear mostly self-consistent, but conflict with each other in fundamental ways. Thus, they should represented by distinct baseSci kb runs. To unify two or more knowledge base runs, as Tiffany0 does above, all subsumed knowledge base runs must be based upon the same baseSci knowledge base run (and thus the same baseEnv knowledge base run). The purpose of this requirement is to dissuade attempting to unify completely distinct views of science.

15.2 Knowledge Base Run Tables

Each knowledge base run keeps track of the prior knowledge base runs which it extends in a table:

1. The official name of the knowledge base run
2. A nickname of the run, which must conform to the naming conventions of an identifier
3. The time and date of creation
4. The time and date of finishing (if applicable)
5. The time and date of publishing (if applicable)
6. The name of the creator/owner
7. Public key of creator/owner
8. A cryptographic hash

Both the official names and the nicknames must be unique within the same table.

Let Mike0, Mike1, Joe0 and Joe1 serve as nicknames in all the appropriate tables. If Mike0 defines the concept "*nucleus*" to mean "*the central, massive part of an atom*", but Joe0 defines "*nucleus*" to mean "*the organelle of a eukaryote cell that contains most of its DNA*", then Tiffany0 may refer to an atomic nucleus as Mike1:Mike0:nucleus, and may refer to a cellular nucleus as Joe1:Joe0:nucleus.

15.3 Knowledge Base Run Linearizations

Knowledge base runs have tables containing baseEnv, baseSci, any other knowledge base runs that they directly build upon, and themselves. However, they also have ***linearizations*** of all prior runs upon which they directly or indirectly build. The purpose of linearizations is to establish sub-sequences of knowledge base runs minimize how often they jump among differing extensions. This is

especially important for method inheritance, which is discussed in the next chapter.

Let knowledge the following knowledge base runs be finished on the following dates:

1. `Joe0` on 2018 May 1
2. `Mike0` on 2018 June 1
3. `Joe1` on 2018 July 1
4. `Mike1` on 2018 August 1

The knowledge base linearization for `Tiffany0` will list the following knowledge base runs in the following order:

1. `baseEnv`
2. `baseSci`
3. `Joe0`
4. `Joe1`
5. `Mike0`
6. `Mike1`
7. `Tiffany0`

The `Joe0`, `Joe1` path will be listed first because `Joe0` had and earlier finish date than `Mike0`. `Joe1` is listed before `Mike0`, even though `Joe1` was finished after `Mike0`, because presumably the jump between `Joe0` to `Joe1` is less abrupt than that between `Joe0` and `Mike0`. The one abrupt jump between `Joe1` to `Mike0` is less jarring than three such jumps imposed by the purely temporal ordering of `Joe0` to `Mike0`, `Mike0` to `Joe1`, and `Joe1` to `Mike1`.

More formally, whenever there is a fork, the linearization follows the knowledge base run with the earliest finish date and time until it joins with some other path (and it *will* join up because all paths must join at `thisRun`, if not earlier). The next path taken is the path with the second earliest finish date and time, *etc*. If there are two or more paths with knowledge base runs with the exact same finishing date and time, then they are ordered by their cryptographic hash in ascending order.

15.4 The Recommend Semantics And Pragmatics of Knowledge Base Run Unification

A knowledge base is largely defined by its ontology. This ontology states what does (and by omission, what does ***not***) exist. The Scienceomatic environment supports Kuhnian “normal science” with its ready support for knowledge base runs to extend ontology at the leaves. Further, unifying two or more runs is easiest when they have extended non-overlapping areas of the ontology.

There is less support for reorganizing the internals of a temporally-inherited ontology. Such an operation is more akin to a “scientific revolution”, and is better addressed by creating a brand new knowledge base from either `baseSci`, or if necessary `baseEnv`.

16. Variables and Flow Control

StructProc can be used as a general-purpose programming language with variables, conditionals, loops and function calls. Conditional and looping structures are implemented by asserting values for attributes special attributes. One *could* do this with `assertX()`, but it makes for needlessly verbose code. Instead it is expected that you will make use of the implicit anonymous construction as detailed in chapters 7 and 8.

14.1 The if-statement: `If`

Constructors are inherited from classes, like `If`. To construct an instance of class `If`, prepend it with a caret (meaning “make an instance of this set only once when this code is read, a.k.a. '*read-time*'”). Place the arguments to the constructor in a comma-separated list delimited by curly braces. Thus,

15. Iterators, Part 1

15.1 The Philosophy of StructProc Iterators

StructProc is for science. Scientists often want several answers to their queries, whether or not they are mutually agreeing or contradictory. Iterators encapsulate the state of a query or search; thus they are an essential part of StructProc.

StructProc also is a language in the artificial intelligence tradition, is designed for artificial intelligence type problems, and assumes the artificial intelligence worldview. For those in artificial intelligence querying is inherently "hard": exponential, or at best NP-complete. Accordingly, StructProc consciously restricts iterators.

Iterators can only be relied upon to support four kinds of operations:

1. Report their current finding (or return `null` if all findings have been exhausted).
2. Be advanced to the next finding.
3. Tell you if they have exhausted all of their findings.
4. Be reset back to the beginning of their query or data-structure.

Note: there is no "go back to the previous finding" operation: the fear being that might involve too much space to cache, or too much time to recompute. (However, you as programmer are free to cache it yourself.)

Contrast the artificial intelligence worldview with that held by computer scientists with backgrounds in data-structures and databases. For them querying is "easy". In general, if your query is worse than O(lg n) then you have chosen the wrong data-structure/algorithm pair, or you have not properly normalized your tables. Thus, iterators through data-structures can do all kinds of fancy things, like go forwards in backwards, with equal agility. Not so for computer scientists with backgrounds in artificial intelligence, and therefore not so for StructProc.

In finer detail, the philosophy of StructProc iterators is that they act as if they always point to the current finding, or to an "end-of query/data-structure" marker. (Though this may be an illusion; their actual implementation may use a lazy, "just-in-time" computation of the current finding.) The conceptual advantage of this are that:

1. At any point in an iterator's life it has one answer for you (even though this may be an illusion, asking for that one answer *the first time* may involve a lot of work), and,
2. Asking the *same iterator* for the *same answer* the *second time* should (ideally) involve much less work. Ideally it should just require a finite sequence of O(1) look ups (assuming the relevant knowledge is still in memory).

To be fair, there are other ways to think about iterators. For example, in Java iterators through linked lists conceptually point in between list items. The Java-approach conceptually simplifies the task of insertion into a linked list: the new item goes in between the two items to which the iterator points. StructProc allows insertion at an iterator-specified position in a `NodeList` instance, but its interface for doing so is not as clean as Java's. However, unlike Java, StructProc iterators support general querying of the knowledge-base, so the "pointed to answer" approach is more applicable.

15.2 The Basics of Iterator Usage

We already have seen iterators used in loops.

```
SP9a1 :) [ 1]  ^Do
[ *[
```

```
  ^VarDecl[* @iter, Iterator *],

  ^For
  [*
    @iter := [Mary,had,a,little,lamb]->dataStruct_iter(),
   !@iter->iter_isAtEnd(),
    stdOut->print(@iter->iter_value())->print(" "),
    @iter->iter_advance()
  *],

  stdOut->printLn(""),

  ^For
  [*
    @iter->iter_reset(),
   !@iter->iter_isAtEnd(),
    stdOut->print(@iter->iter_value())->print(" "),
    @iter->iter_advance()
  *]
]*];
Mary had a little lamb
Mary had a little lamb stdOut
```

The example above demonstrates defining an iterator to iterate over a list, and then resets it to iterate over the list again.

The basic methods of iterators are:

Method	Description
`iter_isAtEnd()`	Returns `true` if the iterator has gone past the last item, or `false` otherwise.
`iter_value()`	Returns the item at the iterator's current location, or `null` if `iter_isAtEnd()` returns `true`.
`iter_key()`	Returns the key at the iterator's current location. In general, this is exactly the same as `iter_value()`, but when iterating through instances of `Map`, `iter_key()` returns the key of a <key,value> pair, while `iter_value()` returns the value. (See section 11.5 on Maps)
`iter_entry()`	Returns the complete derivation of how a value was computed, including the value. For iterators over the ontology, and for iterators through data-structures, this is just the value itself. For them its behavior is identical to that of `iter_value()`.
`iter_advance()`	Advances the iterator to refer to the next item. Returns the item at the iterator's old location.
`iter_reset()`	Resets the iterator to refer to the beginning of its data-structures. Returns

Method	Description
	the iterator itself.

The question "*How do I generate an iterator?*" is as important as "*What can I do with one?*" The table below lists the StructProc architecture-supported iterator-returning expressions to iterate over data-structures and through the ontology.

Through what	Expression which returns iterator:
Items in data-structure `dataStruct`.	`dataStruct->dataStruct_iter()`
Superclasses to which class `class` belongs.	`class->class_superClassIter()`
Subclasses underneath class `class`.	`class->class_subclassIter()`
Classes to which instance `instance` belongs.	`instance->instance_classIter()`
Instances of class `class`.	`class->class_instanceIter()`
Only locally-asserted values of instance `instance` for attribute `attr`.	`instance->instance_localPropIter(attr)`
All values (locally-asserted first, then inherited) of attribute `attr` for instance `instance`.	`instance->instance_propIter(attr)`
Values inherited by instances of class `class` for attribute `attr` that have been subAsserted at `class`.	`class->subProp_iter(attr)`

16. Data-Structures: Vector Lists, Node Lists, Bags and Maps

StructProc is not meant to be a general-purpose programming language like C++, Java or C#. It has limited abilities to let programmers define their own data-structures (beyond, of course, the structures of its property quadruplets). It does have, however, four built-in datatypes: vector lists (instances of `VectorList`), node lists (instances of `NodeList`), bags (instances of `Bag`), and maps (instances of `Map`). StructProc's design philosophy is to have fewer, but more general-purpose data-structures. Thus, bags take the place of set data-structures in other languages, and vector lists take the place of arrays.

16.1 Vector Lists

We have seen compile-time vector lists already. For example, the sole parameter to `Do`'s constructor is a list of commands to do.

```
SP9a1 :) [1]  ^Do
[*[
  stdOut->printLn("A"),
  stdOut->printLn("B"),
  stdOut->printLn("C")
]*];
A
B
C
stdOut
```

In the case above that list is `[ stdOut->printLn("A"), stdOut->printLn("B"), stdOut->printLn("C") ]`.

An empty `VectorList` that is allowed to hold instances of any class may be constructed at compile-time with the usual class constructs of `^VectorList[**]` or `^VectorList`. However, simply enclosing nothing between `[` and `]` is the most concise syntax:

```
SP9a1 :) [1] ^VectorList[**];
[]
SP9a1 :) [1] ^VectorList;
[]
SP9a1 :) [2] [];
[]
```

To limit the compile-time created `VectorList` to only hold instances of a particular class, specify that class as the sole argument to the constructor list, as in `^VectorList[*Integer*]`, or more concisely, `[][*Integer*]`. Attempts to create lists with instances that do not belong to that class, or to insert them in after creation, will result in an `ItemNotInRequiredDomainException`.

```
SP9a1 :) [3] ^VectorList[*Integer*]->dataStruct_insertZ(1);
[1][*Integer*]
SP9a1 :) [4] ^VectorList[*Integer*]->dataStruct_insertZ(1.0);
1. at column 25 of line 5 of the command line is not in class Integer,
therefore dataStruct_insertZ could not be done
SP9a1 :) [5] [][*Integer*]->dataStruct_insertZ(1);
```

```
[1][*Integer*]
SP9a1 :) [6] [][*Integer*]->dataStruct_insertZ(1.0);
1. at column 16 of line 7 of the command line is not in class Integer,
therefore dataStruct_insertZ could not be done
SP9a1 :) [7] [1][*Integer*];
[1][*Integer*]
SP9a1 :) [8] [1.0][*Integer*];
1. at column 5 of line 9 of the command line is not in class Integer, therefore
dataStruct_insertZ could not be done
```

To create a VectorList at runtime, say `new VectorList`, or `new VectorList[*Class*]` to restrict it to only hold members of a given class.

```
SP9a1 :) [1] new VectorList;
[]
SP9a1 :) [1] new VectorList[*Integer*];
[][*Integer*]
SP9a1 :) [2]
```

To retrieve an element from a `VectorList` use the `vList_get(index)` method. `index` should be an integer. If the vector has never been extended by `dataStruct_insertA(item)` then the indices will range from 0 to the list's size minus one. `vList_get(index)` either returns the item at that position (if `index` is in range) or `null` if `index` is out-of-range.

To overwrite an element in a `VectorList` use `vList_didPut(index,item)`. If `index` is in range then this method places `item` in the vector list at position `index`, and returns `true`. However, if `index` is not in range, then the method returns `false` and no other action is taken.

16.2 Methods of Data-structures

Now that we have seen our first data-structure, we can mention the methods of all data-structures.

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

Method	Description
	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. 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.

Method	Description
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.
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)	

The methods of *classes* of data-structures (VectorList, NodeList, Map, Bag and DataStructure), rather than of data-structures themselves are listed below. (They are accessible from the class of the data-structure rather than from individual data-structures themselves. The closest level of organization in a conventional object-oriented language like C++ or Java is static methods.)

Method	Description
dataStruct_getIdenticalSingleton(source)	For all: if source is the first instance of a data-structure that is identical to itself, then (1) source is internally registered, (2) it is marked as immutable (is given property

Method	Description
	<ideasIsImmutableA,true>), and (3) source is returned. However, if another data-structure with the same contents in the same order/mapping/count as source already has been registered, then that already registered data-structure is returned.

16.3 Node Lists

Although both dataStruct_insertA(item) and dataStruct_insertZ(item) are defined on vector lists, vector lists are limited in that insertion cannot be done in the middle. Further, while dataStruct_didRemove(item) is defined on vector lists, it is a O(n) operation (assuming the list is in memory).

Node lists, however, are not restricted in this manner. While node lists cannot do array-style access via vList_get(index) and vList_didPut(index,item) (and vList_numInsertA() is similarly undefined), they can do all the other list methods.

Unlike vector lists, which are built upon extendable arrays of contiguous memory, node lists are built of discrete nodes in memory. Each node list node keeps track of the node before it, the node after it, and the item to which it refers. Additionally, all StructProc node lists have their own nil nodes, which mark the end of the list. (NOTE: the nil nodes mark the end of a list, and are distinct from the null idea used to denote "*No more answers*.")

(Examples)

This next example demonstrates a variety of list manipulations. Note that compile-time node lists begin with [** and end with **].

17. User-Defined Methods

Besides calling predefined methods, StructProc programmers may define their own methods. They are owned by some class or instance of the ontology.

17.1. Introduction

An example of a user-defined method is presented below:

```
Idea
{*
   sub hello.str
   [* compMeth |
        [
          [@name, String]
        ],
        String,
        ^Do
        [*[
          ^Return[* "Hello " ++con @name  *]
        ]*]
   *]
*};
```

The ***computational method*** (or colloquially just "method") is prefaced with `sub`, meaning that although it is owned by `Idea`, the method is for its instances. The alternative to `sub` is `self`, which means that the method is for the instance in which it is defined.

After the `sub` or `self` is the ***method family name*** (in this case `hello.str`).

After the method name is the definition of the method within the `[* *]` pair. First is the keyword `compMeth`. A vertical bar then separates this keyword from the rest of the definition, specifically the parameter list, return type, and code. It is meant to convey "*`hello.str` is a computational method such that it has the parameter list, return type and code given as follows . . .*"

The parameter list is a list of lists. If the method takes no arguments then the parameter list is just the empty list, `[ ]`. However, if arguments are given, then each parameter must have one of the following forms:

```
[@variableName]
```

or, more preferably:

```
[@variableName, VariableDomainClass]
```

or sometimes:

```
[@variableName, VariableDomainClass, nullArgAllowed]
```

All three forms define a new parameter variable (`@variableName`). In the first case, this domain class defaults to `Idea`. In the last two cases, this variable is limited to values in the specified domain class (`VariableDomainClass`). In the very last case the constant `null` is allowed as an argument, even if `null` is not in class `VariableDomainClass`. The term `nullArgAllowed` gives programmers control over whether or not `null` is allowed as an argument (by default it is not if `null`

is not in `VariableDomainClass`). We consider this more elegant than always allowing `null`, or always disallowing it and forcing the user to define two different methods. By default it is disallowed, unless the user specifies otherwise with `nullArgAllowed`. And if they explicitly specify `nullArgAllowed`, they should remember to handle a `null` argument in their method.

The return type may be a class (***or a datatype prototype?***). In the example above it is simply `String`. All methods must return something, so there is no equivalent of `void` as in C, C++ and Java. Methods that would return `void` in C, C++ or Java commonly return `@this` in StructProc9.

The last argument is the code of the method. Generally this code is wrapped in a `^Do[ *[ … ]*]` block, although `^Return[ * .. *]` and `^If[ * .. *]` are also allowed. This code may only use the variables given as arguments, defined within the code itself, or that are the special method variables `@this` (which tells the subject of the method call, the entity to the left of the `->`) or `@these` (telling the class that owns the method).

The use of the vertical bar ("such that") to separate `compMeth` from the rest of the method definition is deliberate. The method definition above could be read as "*`hello.str` is a computation method <u>such</u> <u>that</u> it takes one `String` argument, returns an instance of a `String`, and executes the specified code.*"

The simplest way to call a method is to give a ***<u>subject</u>*** that is an instance of the class in which the method resides, the arrow `->` , the method family name, and the desired arguments of the correct domains between a parenthesis pair.

```
1->hello.str("Joe");
```

Here, the `Integer` value `1` (or anything else in the knowledge base) is an instance of `Idea` and serves as the subject of the call. This `1` will reappear in the body of the method in the variable `@this`. This method has one parameter, and in this example the variable `@name` will take on value `"Joe"`.

It is illegal to name a method the same name as a standard, predefined method. Doing so would be a security risk, analogous to an operating system allowing arbitrary processes to change its interrupt table.

17.2 Method Linking

The linker must deal with 3 issues:

1. Finding the best matching method in the hierarchy given the subject, `@thisSubj`. For example, if the method `isOdd()` exists under both `Integer` and `Idea`, then the call `1->isOdd()` should link with the one under `Integer`.
2. Finding the best matching method in the hierarchy given the arguments. For example, two methods named `print()` exist under `OutputStream`, but one takes a `Rational` argument (`print(Rational)`) while the other takes a `String` argument (`print(String)`) then the call `stdOut->print(1)` should link with `print(Rational)`.
3. Finding the best matching method in the knowledge base given the time the method was defined, and the desires of the user. For example, a knowledge base has three runs created on 2018-April-15, 2018-May-15 and 2018-June-15. A method `f()` under `Idea` was
   - defined in 2018-April-15,
   - called in code on 2018-May-15,
   - redefined on 2018-June-15.

> Under `linkForward` the call written in 2018-May-15 would link with the version of `f()` then existing: from 2018-April-15. Under `linkBackward` the call written in 2018-May-15 would link with the most recent version of `f()`: from 2018-June-15.

The first two of these issues are shared by many object-oriented languages, like C++ and Java. The last issue, however, is unique to monotonic knowledge base languages like sp9a.

The first issue is addressed in a manner similar to that used by C++. If there are two or more contending method definitions, class search is executed from the instance `@thisSubj`. The class that has a matching method is called.

The second issue is addressed with a programmer-defined extension. All method prototypes should be given a unique extension by the programmer, like `hello.str` for a method that takes a string as its sole argument. (A warning will result if the extension is missing.) Much like file system entry names, the extension portion of the name follows the period. The ***fullname*** of a method includes its extension.

The system enforces the uniqueness of the fullnames of methods: if ever the user tries to violate it by attempting to define the following:

```
class  Whatever
{*
   sub compMeth
   [* hello.str |
        String,
        [[@name, String]],
        ^Do
        [*[
          ^Return[* "Hello " ++con @name  *]
        ]*]
   *]
*};

class WhateverElse
{*
   sub compMeth
   [* hello.str |
        Integer,
        [[@int, Integer]],
        ^Do
        [*[
          ^Return[* 5 + @int  *]
        ]*]
   *]
*};
```

then an error will result, even though the methods are in two different classes. The second method is trying to repurpose `hello.str` to a different prototype, one that takes an `Integer` and returns an `Integer` instead of taking a `String` and returning a `String`.

Method calls should use these extensions. Warnings will result when they are not used, even when there is no ambiguity. If there is ambiguity then an error will result and linking will fail. This should be resolved by specifying extensions.

Conflicts may arise when combining two different knowledge bases. In this case, the combining algorithm may rename one or both prototype extensions.

The third issue is dealt by the giving a link specifier and/or a knowledge base run. One example is:

```
subject->kbRun:className::methodName.extension(methodArgs)
```

(Note the double-colon used after, and only after, the given class.) This form needs no link specifier because the user is explicitly specifying the knowledge base run and the class. The only requirements are that:

1. The knowledge base run actual defines the named method at the given class, and,
2. That the subject be an instance of the given class.

More generally one may have sp9a find and link with the appropriate method over knowledge base runs (*temporal-inheritance*), and in the ontology (*ontological inheritance*). The general form for this is:

```
subject->linkSpecifier:kbRun:methodName.extension(methodArgs)
```

(There should also be a form for calling a method with both a subject and a justification, perhaps something like

```
(subj,just)->linkSpecifier:kbRun:methodName.extension(methodArgs)
```

if it proves unambiguous.)

The link specifiers are:

1. `linkOnlyWith`: Means "*link only with the knowledge base run given next*". This specifier must be used with a named knowledge base run. If *only* a knowledge base run is given then this is the link option that is used.
2. `linkForward`: Means "*link with the first knowledge base run with a matching method, starting with the earliest run, `baseEnv`, and going forward in time*". This option requires no knowledge base run specified. If a knowledge base run is given, then it is ignored. Without a knowledge base run, the knowledge base run of the method call is taken as `null`, which is the suggested value to be used if a knowledge base run *must* be given.
3. `linkBackward`: Means "*link with the first knowledge base run with a matching method, starting with the current run, `thisRun`, and going back in time*". This approach has the property of always linking with the most modern definition of a method. If neither link specifier nor knowledge base run is explicitly given, then this is the default link specification.
   As with `linkForward`, this option requires no knowledge base run specified. If a knowledge base run is given, then it is ignored. Without a knowledge base run, the knowledge base run of the method call is taken as `null`, which is the suggested value to be used if a knowledge base run *must* be given.
4. `linkForwardUntil`: Means "*Starting with the first knowledge base run `baseEnv` and moving forward in time until (and including) the specified knowledge base run, link with the first knowledge base run with a matching method*". This option requires that a knowledge base run be explicitly specified.
5. `linkBackwardUntil`: Means "*Starting with the current knowledge base run `thisRun` and moving back in time until (and including) the specified knowledge base run, link with the first knowledge base run with a matching method*". This option requires that a knowledge base run be explicitly specified.

6. `linkBackwardSince`: Means "*Starting with the specified knowledge base run and moving back until (and including)* `baseEnv`*, link with the first knowledge base run with a matching method*". This option requires that a knowledge base run be explicitly specified.
7. `linkForwardSince`: Means "*Starting with the specified knowledge base run and moving forward until (and including)* `thisRun`*, link with the first knowledge base run with a matching method*". This option requires that a knowledge base run be explicitly specified.

The distinctions among `linkForwardUntil`, `linkBackwardUntil`, `linkBackwardSince` and `linkForwardSince` may are shown in the diagram below: "`forward`" vs. "`backward`" tells direction in time, while "`until`" vs. "`since`" tells starting point ("`until`" means start at the endpoints, while "`since`" means start at the specified knowledge base run).

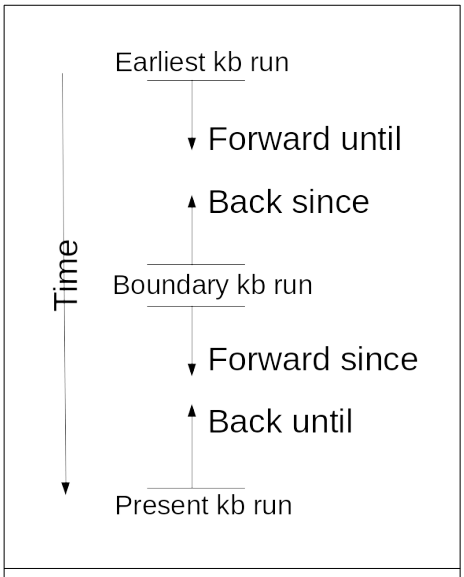

Figure 13.1: Until vs. Since, and Forward vs. Backward

Either, or both, the link specification and knowledge base run specification may be given at run-time as variables.

Any user-defined method temporally-inherited at a class may be undefined in the current active knowledge base run by ***masking*** it. This is done like the following:

```
class  Whatever
{*
  sub compMeth
  [* hello.str | masked *]
*};
```

Here, `hello.str()` is taken to no longer exist in this knowledge base run at class `Whatever`. Using `linkForward` will still find the first `hello.str()` method defined at `Whatever`, but using `linkBackward` for an instance of `Whatever` will invoke a `hello.str()` defined in a superclass of `Whatever` (if such a method exists).

This example should illustrate the interplay between ontological-inheritance and temporal-

inheritance. Let two classes exist: `Base`, and, a class that is derived from it, `Derived`. Let `derivedInstance` be an instance of `Derived`.

- If in knowledge base run `run0`, a method `foo()` is defined in `Base`, then `linkBackward:derivedInstance->foo()` will run `run0:Base::foo()`, as it is the only contender.
- If `run1` extends `run0`, and it defines `Derived::foo()`, then `linkBackward:derivedInstance->foo()` will run `run1:Derived::foo()` because it belongs to a more proximate class to `derivedInstance`.
- If `run2` extends `run1`, and it re-defines `Derived::foo()`, then `linkBackward:derivedInstance->foo()` will run `run2:Derived::foo()` because it belongs to a later knowledge base than `run1:Derived::foo()`.
- If `run3` extends `run2`, and it masks `Derived::foo()`, then `linkBackward:derivedInstance->foo()` will run `run0:Base::foo()` because all methods `foo()` at `Derived` have been masked.
- If `run4` extends `run3`, and it defines `Derived::foo()`, then `linkBackward:derivedInstance->foo()` will run `run4:Derived::foo()` because now the mask itself has been masked by a new method.

20. Domain-dependent Mathematical Operations

20.1 Introduction

So far the numbers we have seen, instances of `Integer`, `Rational`, `Real` and `Complex`, have been nonannotated. 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:

21. Scienceomatic Environmental Support

21.1 Introduction

Although Struct Proc may be used as a stand-alone programming environment, it was designed to be an integral part of the larger Scienceomatic knowledge representation and reasoning environment. Within this environment it was designed to play a dominant role in the representation of *symbolic* knowledge (as opposed to *procedural* knowledge, like which statistical tests should be applied when, or media data, like the storage of images, videos or audio files). It was also meant to meaningful, but incomplete, role in *reasoning* with symbolic knowledge (*e.g.* it is expected that the Scienceomatic environment will call on more specialized programs to solve non-trivial equations, do specialized linear algebra problems, apply statistical tests, *etc*.)

Struct Proc supports this larger environment with a variety of methods.

`obtainStdPropertiesAndArbitraryAttrs()` of `Idea`: The purpose of this method is to return the readily-accessible properties of the subject on which the method is called. It may be run on any ontological entity and has no parameters. It returns a `VectorList` instance of `VectorList` instances. Each inner VectorList instance has 3 members: the first is the attribute, the second is the first listed value of that attribute, and the third is the justification corresponding to the attribute and justification. If no value is listed then the holder value `unknown` is given. The "justification" `byNonsense` corresponds to any `unknown` or `null` value.

```
SP9a1 :) [ 1] 1->obtainStdPropertiesAndArbitraryAttrs();
[[ instanceOf,Integer,byDefinition],
[ mayBeDestroyedA,false,byDefinition],
[ isImmutableA,true,byDefinition]]
SP9a1 :) [ 1] "Greetings"->obtainStdPropertiesAndArbitraryAttrs();
[[ instanceOf,String,byDefinition],
[ mayBeDestroyedA,false,byDefinition],
[ isImmutableA,true,byDefinition]]
```

`getInfoForNavigationPage()` of `Idea`: The purpose of this method is to return the ontological neighbors of `@thisSubj`, to facilitate navigation through the knowledge base. It may be run on any ontological entity and has no parameters. It returns a `VectorList` instance of with 5 arguments. The first is the just the `@thisSubj`, the thing being described. The second is the first listed class of @thisSubj. The third is the first listed superclass of `@thisSubj`, or is `null` if `@thisSubj` is not a class. The fourth is a `VectorList` of immediate subclasses of `@thisSubj`, or is an empty `VectorList` if `@thisSubj` is not a class. The fifth and last is a `VectorList` of immediate instances of `@thisSubj`, or is an empty `VectorList` if `@thisSubj` is not a class.

```
SP9a1 :) [ 1] 1->getInfoForNavigationPage();
^SOMNavigationPage[*1,Integer,null,[],[]*]
SP9a1 :) [ 1] Idea->getInfoForNavigationPage();
[]
```