

Semantics of Mathematical Writing

Joe Corneli*

Spring 2004

Abstract

I translated 100 definitions from the PlanetMath encyclopedia into a LISP-like language, here called hyperreal, later renamed hcode and developed further.

How to read this document. The main part of this document consists of the 100 shortest definitions and theorems from PlanetMath and my translations of these definitions into a representation language for mathematics that I'm developing. The document can be used as a "dual text" to learn this new mathematical language. Since the language is still in the development stages, you can also use the document as a basis for critiquing the choices I've made with the new language – please let me know your suggestions if you have any. The draft specification for this language is given in an appendix.

Introduction.

This file illustrates the way an "expert" (me) understands mathematical definitions. But there are some caveats to go along with that statement. First of all, I am coming at the material from a knowledge engineering perspective at least as much as and probably more than from a mathematician's perspective. This bias notwithstanding, I haven't always captured all of the information I find in the definitions presented in the original text – partially because the representation language I'm working with is still in a formative stage. This is particularly true for *amathematical* aspects of the representation – synonyms, "concepts", and abbreviations, for example. I've put some effort into writing something down when I encounter such stuff, but mostly my contributions to this document will consist of cold hard mathematics written in a new way. Finally, let me also point out that despite my claiming to have "expertise", I am not an expert in all of the mathematical areas that the definitions here come from. When I have encountered particular difficulties in a certain translation (whether because of a lack of mathematical knowledge on my part, a lack of clarity in the source,

*The content of this file is available under your choice of the GNU Free Documentation License (FDL) and the GNU General Public License (GPL). If the FDL is selected, be advised that this document has no Invariant Sections, no Front-Cover Texts, and and no Back-Cover Texts.

or questions at the knowledge engineering level), I have made some additional notes describing the problem.

If you feel like being imaginative, you could imagine that the translation had been prepared by a machine and that these comments were its auto-generated feedback on its translation process. This is of course a fiction, and it will quickly be seen to be a silly one. Nevertheless, my hope is that David Kim and I will be able to write/understand the rules that go into making these translations well enough to formalize them in a computer program.

Before proceeding, I will make a few comments about the *representation language* and the *process of building representations*. Some of the problems are fairly complex. The issues that we must address include:

1. How to introduce new notation?
2. How to extract pieces of a definition? (E.g. the group operation that is associated with a group G – can we just call it $*$?)
3. How to deal with global objects (like \mathbb{R} or \mathbb{C})?
4. What clues do we use to decide when we are looking at a theorem and when we are looking at a definition (or an example or something else)?
5. What syntax will we use to represent a map from one space to another?
6. What will a definition or a theorem take as an argument?
7. What will a definition or a theorem return?
8. What data structures will be available?
9. How do we distinguish between a definition and a description?
10. What about things that are not math per se at all?
11. What is some other related stuff I can read instead or in addition?
12. Why make a new language for mathematics on the computer?
13. How to reconcile this language with formal mathematics?
14. What is the new language supposed to be called?

The treatment of these issues that we give here sometimes will summarize or expand upon things that are brought up in the notes on translation that appear in the body of this text.

The difference between *assertions* and *instantiations*. In a typical defn, I will make a list of assertions – the objects relevant to the definition satisfy some collection of predicates. However, this convention means that it is not correct to introduce *new notation* with the same style. I noticed this in particular when I was looking at the definition of prime-ring.

```
(defn prime-ring (R)
  (ring R)
  (let ((Z (zero-ideal-of R)))
    (prime-ideal Z R)))
```

Had we not been paying such careful attention to the difference between assertions and instantiations, this might have been written

```
(defn prime-ring (R)
  (ring R)
  (zero-ideal Z R)
  (prime-ideal Z R))
```

The reason for not doing things this way is to be consistent with the style of programming used in typical LISP: one does not talk about variables that have not been instantiated. Put another way, if we want the convenience of having everything inside of a `defn` be inside of an `and`, we have to be careful to make a distinction between the things that we are *saying* are true, like `(zero-ideal Z R)` in this case, and the things we are *testing* for truth, like `(prime-ideal Z R)`. Even if we later change the representation language to allow somewhat more lax definitions, we will have to be careful about this distinction internally. At this point in the game, it is far better to be explicit. On the other hand, there are a few efficient ways to write things are so natural that we really can't do without them: for example, you should be able to introduce new notation inside of an `exists` or a `forall` form. More comments on things like this in our spec.

Extraction. In the example given above, we are using the more LISP-like `(zero-ideal-of R)` instead of a more KM-like `(the zero-ideal of R)`. It isn't totally clear to me what the best way to extract the zero ideal of R is. I'm currently inspired by this somewhat odd mixture of KM and LISP styles. But I think I'll probably try to stick to a LISP-like style as much as possible. On the other hand, KM has some very nice idioms for dealing with selection of *one* of *many* things ("a foo of quux") as opposed to *one* of *one* things ("the foo of quux"). So I'm not going to rule out a switch to a more KM-like syntax for these sorts of things.

The issue of extraction is of quite general importance. Our general idea is that within any top-level form, certain objects will add some default information to the name-space. For example, if we know G is a group then for two elements of G , g and h , we may be able to talk about gh without explicitly stating what the multiplication is (unless of course there is liable to be some confusion). But this doesn't tell us exactly what notational system to use when talking about a group's operation or a ring's zero ideal.

One approach might be related to considerations about which parts of a `defn` are optional; for example, `(group G)` should say let G be a group. But `(group G foo)` would say that G is a group with operation *foo*. If the group operation isn't supplied explicitly, it would be auto-generated as needed. Maybe we need to be explicit about this, and say that the concept of "group" has an associated definition, namely "group operation of", that you could use to explicitly write something like `(let ((* (group-operation-of G)) ...)`. If we already knew that the group operation was called *foo* (and this was the only viable operation on G that was on the table), we would then have `(eq * foo)`. If the set G had a few different group operations

defined on it, we might get into some trouble with this approach; anyway, it becomes more complicated.

Being able to get the group operation from context (respecting the scope of the forms we use) would just be shorthand for a more explicit treatment like the one used here. So far, in the code that I've written, I've used an *implicit* approach to this issue; that may change as the code evolves.

Mathematical spaces and other global variables So far we don't really have a convention for dealing with global variables like \mathbb{R} . Eventually there should be a "standard library" of functions and spaces, etc. Currently the usage is not terribly consistent – but I think for the most part these things are treated as global variables.

What... is this? One simple cue is that things that take an

If... then...

form are usually theorems. Things that take a

... is a ...

or

... is simply a ...

form are usually definitions. One other really super cue is that text that is being defined is often *emphasized*. (But this visual cue is not always available, and sometimes things that are emphasized are *not* terms that are being defined.) We should take a good look at the rules that were used by Don Simon in his thesis, since he nicely catalogs many of the important modifiers (like "simply") and many of the important forms. Anyway, we should make our own list of cues.

Syntax of maps. It is easy to write $(\text{map } f \ X \ Y)$ to denote the fact that that f is a map from X to Y . Similarly with other kinds of maps (functions, functors, differentiable maps, etc.). So I guess this is the *de facto* standard for now. We will also use $(\text{function } f \ X)$ to denote a function defined on X – because *function* in math typically means a map to \mathbb{R} or \mathbb{C} , the range of the function should be determined from context (defaults to \mathbb{R} , say).

Arguments. Complementing the comment that states that we don't want to use symbols that aren't defined is the fact that we sometimes expect to have symbols coming in from "outside" that we will have to deal with. This is relevant for example to *testing* whether something satisfies a certain definition. If we want to know whether G is a group, then we may want to write something like $(\text{is-a-group-p } G)$. In fact, we will just write $(\text{group } G)$, for reasons that will become clear later if they have not already; the definition of a group will be that it passes the test we have prepared for being a group. (It is easy to see how this relates to the problem of *extraction* mentioned earlier.) On the other hand, we will often want to use our definitions to *generate* an arbitrary group, or whatever other object we're interested in, for use

in a theorem (as in, “Let G be a group...”). But theorem themselves should also take arguments – the arguments that would appear in natural language should give us a clue about what arguments the theorem will take (as in, “By so-and-so’s theorem applied to such-and-such...”).

Return types. Along with arguments, we want to know return types. As mentioned above, on one level, we just want to use the definition of a group to test whether something is a group. On the other hand, we also want to use the same definition to generate an arbitrary object that is a group. Similarly with our other definitions. This is relevant to the issue of *extraction* mentioned above. It is also relevant for humans reading the code: humans need to easily be able to see what is being defined. Well, in fact, a lot of this information can be obtained from the argument list. If there is just one argument, that is what is being defined. However, there is often more than one argument, so we need to be more clever than that. This is also relevant for theorems: in general, we want the *last* form inside a *defthm* form to be the conclusion of the theorem, and the rest to be the hypotheses. (This is somewhat sloppy short-hand for putting all of the hypotheses inside of the first bit of an `(if (and ...) ...)` – it isn’t actually clear that such a shorthand is justified, since we sometimes have theorems of the form `(iff (and ...) ...)`, so we may have to edit the *defthm*’s to be more explicit.)

Definitions versus descriptions. I’m thinking of the idiom that runs as follows:

If A then we say B is a C .

In other words, we *aren’t* exactly saying that B is defined to be C – just being a C is one property of B . If a man has a membership in the Protective Brotherhood of Elks, then we say he is an *Elk* – but of course this doesn’t mean that he is *defined* to be an Elk (it would just be part of a bigger definition). However, we can typically extract from this sort of statement some actual definition – here we can extract the *definition of an Elk* – namely, that an Elk is someone who belongs to the Protective Brotherhood of Elks. Deciding how to turn a given description into an appropriate definition can be difficult (see for example Entry 64, which includes a definition of a q -skew polynomial ring created according to a pattern like the one described here).

Non-math. Although lots of mathematical writing falls into some part of the definition-theorem-proof cycle, there are other things that do not (explanations, synonyms, historical or biographical excursions, etc.) What should we do in terms of representing these more complex (or complexifying) ideas? (What should we leave out?) I think that the right way to handle this would be to keep the mathematical entries in *slots* that are part of a more general representation language (like KM). This practice hasn’t been adhered to in the current document; mostly I’ve been concerned with hammering out the mathematical representations, but once the details here are finished, I hope to come back to the non-mathematical parts of mathematical text.

Data structures. General mathematical data structures should be easy to define. Sets, lists, and numbers should be built in types. Truth and falsity will be handled the same way they are in typical LISP.

Other stuff. Aaron Krowne's \TeX Object Model (TOM) may be worth taking a look at. Don Simon's thesis is definitely worth taking a look at. HOL, Coq are well developed systems that are somewhat similar to the one being proposed here. At some point we should give a more detailed comparison between our system and these other extant systems. Nqthm/ACL2 are LISP-based systems (also developed at UT), but they are not higher order; still, there is likely to be a pretty big overlap between them and the system we are proposing.

Why this new language? We wanted something we could easily tweak at the basic levels if things weren't going well. We needed something that was very near (at least in our view) to math as it is written by day-to-day users. And finally, LISP has taught us to love symbolic expressions (i.e. things in parentheses); the s-exp idiom makes possible some efficiencies when coding that the other higher-order math-on-computer systems lack.

How to reconcile this language with more formal approaches? I am planning to write an Appendix that will outline the formal aspects of this language (based on Morse's *Set Theory*).

The name of the language. The budding representation language needs a name, and since there aren't any other candidate names on the table, I'm going to call it hyperreal. This way, if I ever do get around to making the Hyperreal Dictionary of Mathematics using this system, we will have another stellar pun on our hands. (But if a better name comes along, maybe I'll take that one up instead.)

TODO.

From first draft of this. At some point `defthm` should take arguments: the "top-level" symbols that are used in the theorem, so that later we could say "by such-and-such a theorem applied to X, Y, and Z, we have P and Q." But for now, theorems take no arguments (this may be fixed before long).

1 NullTree

A *null tree* is simply a tree with zero nodes.

```
(defn null-tree (T)
  (tree T)
  (eq (card (nodes T)) 0))
```

2 EuclidsLemma

If $a|bc$, with $\gcd(a, b) = 1$, then $a|c$.

```
(defthm euclids-lemma (a b c)
  (int a b c)
  (divides a bc)
  (eq (gcd a b) 1)
  (divides a c))
```

3 Hexagon

An *hexagon* is a 6-sided polygon.

```
(defn hexagon (X)
  (polygon X)
  (eq (card (sides X)) 6))
```

4 Residual

In a topological space X , a set A is called residual if A is second category and $X \setminus A$ is first category.

```
(defn residual-set (A)
  (topological-space X)
  (subset A X)
  (second-category A)
  (first-category (setminus X A)))
```

5 Endofunctor

Given a category \mathcal{C} , an *endofunctor* is a functor $T : \mathcal{C} \rightarrow \mathcal{C}$.

```
(defn endofunctor (T)
  (category C)
  (functor T :from C :to C))
```

6 GaloisExtension

A field extension is *Galois* if it is normal and separable.

```
(defn galois-field-extension (X)
  (field-extension X)
  (normal X)
  (separable X))
```

7 MeasureZero

If (X, M, μ) is a measure space, and $A \in M$, then A is said to be of *measure zero* if $\mu(A) = 0$.

```
(defn measure-zero (A X M mu)
  (measure-space X M mu)
  (elt A M)
  (eq (mu A) 0))
```

Notes. Here in is A that is of *measure zero*, but the definition applies to all the other things too. See the comments about return types in the Introduction.

8 RmsError

Short for “root mean square error”: the estimator of the standard deviation.

```
(defabbreviation rms ()
  (short-for 'root-mean-square-error))

(defconcept root-mean-square-error ()
  "estimator of standard deviation")
```

Notes. This RMS entry breaks into two definitions, neither of which is a formal math definition. What should we do in terms of representing these more complex ideas?

9 Pseudograph

A *pseudograph* is a graph that allows both parallel edges and loops.

```
(defn pseudograph (G)
  (graph G)
  (allows G 'parallel-edges 'loops))
```

Notes. Well, maybe. Part of the problem with this definition is that when they say “a graph that allows bla bla bla” it is already implied that the class of *graphs* allows bla bla bla, otherwise it would be impossible to instantiate so-called pseudograph according to this definition. (Compare the definition of “multiset”, which follows a similar pattern.)

10 Disjoint

Two sets X and Y are *disjoint* if their intersection $X \cap Y$ is the empty set.

```
(defn disjoint (X Y)
  (set X Y)
  (eq (intersection X Y) empty-set))
```

11 Separable

A topological space is said to be *separable* if it has a countable dense subset.

```
(defn separable (X)
  (topological-space X)
  (exists Y :st (and (subset Y X)
                    (countable Y)
                    (dense Y X))))
```

12 Successor

Given a set S , the *successor* of S is the set $S \cup \{S\}$. One often denotes the successor of S by S' .

```
(defn successor (X S)
  (set S)
  (eq X (union S (setof S)))
  (has-notation X S'))
```

Notes. This is kind of a mixture of KM and LISP style. Probably not how we really want to write; we want to keep the LISP style self-contained, then embed it in something with a KM style. But we haven't really developed any formal rules for doing this yet.

13 GeneralType

A variety is said to be of *general type* if its Kodaira dimension equals its dimension.

```
(defn variety-of-general-type (X)
  (variety X)
  (eq (kodaira-dimension X)
      (dimension X)))
```

14 WilsonsTheorem

Wilson's theorem states that

$$(p - 1)! \equiv -1 \pmod{p}$$

for prime numbers p .

```
(defthm wilsons-theorem (p)
  (prime p)
  (cong (factorial (- p 1))
        -1
        :mod p))
```

Notes. Saying that something is a prime automatically says that the thing is a number... or at least this is the default. (There are other kinds of objects that can be "prime" but they should not be the default.)

15 HeineBorelTheorem

A subset A of \mathbb{R}^n is compact if and only if A is closed and bounded.

```
(defthm heine-borel-theorem (A)
  (subset A Rn)
  (iff (compact A)
       (and (closed A)
            (bounded A))))
```

Notes. See comments on *spaces* in the introduction.

16 Lindelof

A topological space X is said to be *Lindelöf* if every open cover has a countable subcover.

```
(defn lindelöf (X)
  (topological-space X)
  (forall A (if (open-cover A X)
                (exists B (and (subcover B A X)
                              (countable B))))))
```

Notes. The `:st` keyword can be left off when there are just two arguments to `forall` or `exists`. Maybe it should be possible to leave it off all the time, but I think that when you start putting more things into these forms (multiple variables, an `:in` keyword) then things start to get so complicated as to require the `:st`.

17 AffineSpace

Affine space of dimension n over a field k is simply the set of ordered n -tuples k^n .

```
(defn affine-space (n k)
  (integer n)
  (field k)
  (setof x :st (elt x (cartesian-product n k))))

(defn cartesian-product (n X)
  (integer n)
  (set X)
  (setof (sub x 1) (sub x 2) &c (sub x n)
    :st (forall i :from 1
      :to n
      :st (elt (sub x i) X))))

(defsyn 'tuple 'cartesian-product)
```

Notes. The definition of `cartesian-product` was not extracted from the definition above! I just wrote it down because it seemed useful (now can be part of the “standard library”). It would probably be more efficient to allow

```
(defn cartesian-product (n X)
  (integer n)
  (set X)
  (setof (sub x i) :for i
    :from 1
    :to n
    :st (elt (sub x i) X))))
```

But in fact, even if this is equivalent, it seems harder to read. (And the `:for` token isn't even defined.) So, this “efficient” variant doesn't seem like a very good thing to allow.

18 HilbertsBasisTheorem

Let R be a right (left) noetherian ring. Then $R[x]$ is also right (left) Noetherian.

```
(defthm hilberts-basis-theorem ()
  (let ((t :oneof (right left)))
    (noetherian-ring R :orientation t)
    (noetherian-ring (adjoin R x) :orientation t)))
```

Notes. There is a knowledge gap here (I don't remember much about $R[x]$ and some knowledge about this thing would have to be built into the parsing system for it to produce nice output.)

19 IsomorphicGroups

Two groups $(X_1, *_1)$ and $(X_2, *_2)$ are said to be *isomorphic* if there is a group isomorphism $f: X_1 \rightarrow X_2$.

```
(defn isomorphic (G G')
  (group G)
  (group G')
  (exists f :st (and (map f G G')
                     (group-isomorphism f G G'))))
```

Notes. The unused symbols $*_1$ and $*_2$ seem to be sort of poor form, but its understandable (though not standard) that the original author wanted to use them. However, we don't have a use for them. Also note that the switch to using G and G' instead of X_1 and X_2 is just a “twitch” on my part. It could go either way.

20 NumberField

A field which is a finite extension of \mathbb{Q} , the rational numbers, is called a *number field*.

```
(defn number-field (X)
  (field X)
  (rationals Q)
  (finite-extension X Q))
```

Notes. The order of arguments should be “like in speech” – since this is somewhat arbitrary, `finite-extension` should have a doc-string that reads something like “ X is a finite extension of Y if...”

21 ProperSubset

Let S be a set and let $X \subset S$ be a subset. We say X is a *proper subset* of S if $X \neq S$.

```
(defn proper-subset (S X)
  (set S)
  (subset X S)
  (neq X S))
```

Notes. Here the order of arguments to `proper-subset` comes from the convention that we want to have arguments appear in the order that they are used (here S before X). This may or may not conflict with the other convention used above.

22 DigitalSearchTree

A *digital search tree* is a tree which stores strings internally so that there is no need for extra leaf nodes to store the strings.

Notes. Sorry, I can't quite figure this one out. This doesn't seem to be a math definition per se.

23 FermatsLittleTheorem

If $a, p \in \mathbb{Z}$ with p a prime and $p \nmid a$, then $a^{p-1} \equiv 1 \pmod{p}$.

```
(defthm fermats-little-theorem (a p)
  (integer a)
  (prime p)
  (not (divides p a))
  (cong (expt a (- p 1)) 1 :mod p))
```

Notes. We have to be able to deal *syntactically* with extra/unfamiliar symbols like \nmid . Also, this theorem is an example of the need to do something about the order of entries to a theorem environment – probably shift to using an if... then... format all the time. On the other hand, it is kind of nice to make that built in – and if “iff” theorems are the only exceptions to the order we like, then they aren't really an exception, since a single “iff” form would be the last form that appears in the defthm.

24 LemoineCircle

If through the Lemoine point of a triangle are drawn parallels to the sides, the six points where these intersect the circle lie all on a same circle. This circle is called the *Lemoine circle* of the triangle

```
(defn lemoine-circle ()
  (triangle T)
  (lemoine-point p T)
  (let ((line1 (line :through p :parallel (side 1 T)))
        (line2 (line :through p :parallel (side 2 T)))
        (line3 (line :through p :parallel (side 3 T))))
    (let ((point-a (intersection line1 (side 2 T)))
          (point-b (intersection line1 (side 3 T)))
          (point-c (intersection line2 (side 1 T)))
          (point-d (intersection line2 (side 3 T)))
          (point-e (intersection line3 (side 1 T)))
          (point-f (intersection line3 (side 2 T))))
      (circle :containing point-a
              point-b
              point-c
              point-d
              point-e
              point-f))))
```

Notes. First of all, there seems to be a mistake here: what circle is being talked about? Do they perhaps mean *triangle*? That's what I assumed to make my translation. I should remember to submit a correction to the object's owner on PM.

The translation itself requires some further geometric knowledge (to specify the definitions of point-a – point-e) and also requires a proof to specify the circle containing these points (i.e. to demonstrate that there is such a circle). This is an example of why might need to have a reasoning engine as part of any robust translation-to-hyperreal-format system.

25 HamiltonianPath

Let G be a graph. A path on G that includes every vertex exactly once is called a *hamiltonian path*.

```
(defn hamiltonian-path (G P)
  (graph G)
  (path P G)
  (forall v :in (vertices G) :st (appears-once-in v (vertices P))))
```

Notes. `appears-once-in` is kind of like `elt`, but it applies to lists and *not* to sets.

26 BalancedTree

A *balanced tree* is a rooted tree where each subtree of the root has an equal number of nodes (or as near as possible). For an example, see binary tree.

```
(defn balanced-tree (T)
  (rooted-tree T)
  (forall A :st (if (and (subtree A (root T))
                        (subtree B (root T)))
                    (eq (card (nodes A))
                        (card (nodes B))))))
```

Notes. I don't know how to code "or as near as possible." I also don't know exactly how to code up the "for an example, see..." bit (and I'm not sure if they meant that every binary tree is an example, or that some specific example is made available as part of the definition of binary tree). Probably I should ask for clarification on these points.

27 CardinalityOfACountableUnion

Let C be a countable collection of countable sets. Then $\cup C$ is countable.

```
(defthm countable-union-of-countable-sets-is-countable (X)
  (set X)
  (forall x :in X :st (and (set X)
                           (countable X)))
  (countable (union x :in X)))
```

Notes. The expression $\cup C$ has to be dealt with syntactically – it should produce $(\text{union } c :in C)$ – which has been translated to $(\text{union } x :in X)$ here just because I tend to prefer to X 's as much as possible!

28 ProperIdeal

Suppose R is a ring and I is an ideal of R . We say that I is a *proper ideal* if I is not equal to R .

```
(defn proper-ideal (R I)
  (ring R)
  (ideal I R)
  (not (eq I R)))
```

29 Countable

A set S is countable if there exists a bijection between S and some subset of \mathbb{N} .
All finite sets are countable.

```
(defn countable (S)
  (set S)
  (natural-numbers N)
  (exists M :st (and (subset M N)
                    (exists f :st (bijection f S M))))))
```

30 CrossingNumber

The *crossing number* $cr(G)$ of a graph G is the minimal number of crossings among all embeddings of G in the plane.

```
(defn crossing-number (G)
  (graph G)
  (euclidean-plane R2)
  (let ((X (setof (image G :under f)
                 :st (and (map f G R2)
                         (embedding f))))
        (min (card (crossings x)) :st (elt x X))))
```

Notes. This is one place where I made a somewhat non-trivial hack to the input: `cr` was defined in a header file for the input, so I made a the definition here with the \LaTeX command `\def`. In a more general setting, we will want to have a method for parsing new commands in the \LaTeX files and adding them to the system as we go.

31 Coprime

Two integers a, b are *coprime* if their greatest common divisor is 1. It is also said that a, b are *relatively prime*.

```
(defn coprime (a b)
  (integer a b)
  (eq (greatest-common-divisor a b)
      1))

(defsyn 'coprime 'relatively-prime)
```

32 CancellationRing

A ring R is a *cancellation ring* if for all $a, b \in R$, if $a \cdot b = 0$ then either $a = 0$ or $b = 0$.

```
(defn cancellation-ring (R)
  (ring R)
  (forall a b :in R :st (if (eq (* a b) 0)
                            (or (eq a 0)
                                (eq b 0))))))
```

33 FinitelyGeneratedRModule

A module X over a ring R is said to be *finitely generated* if there is a finite $Y \subseteq X$ such that Y generates X .

```
(defn finitely-generated-module (X R)
  (module X R)
  (exists Y (and (subset Y X)
                 (generates Y X))))
```

Notes. It isn't clear to me whether Y should be a module or just a set; looks like just a set to me.

34 AlgebraicNumberField

A field $K \subseteq \mathbb{C}$ is called an algebraic number field if its dimension over \mathbb{Q} is finite.

```
(defn algebraic-number-field (K)
  (subset K C)
  (field K)
  (rationals Q)
  (finite (dimension K Q)))
```

Notes. I wouldn't usually use us extra tokens unless I need to... but nevertheless, an earlier draft of this entry looked like this:

```
(defn algebraic-number-field (K)
  (subset K C)
  (field K)
  (rationals Q)
  (finite (dimension K :over Q)))
```

Such things may be helpful mnemonics, but they should probably be avoided in code. On the other hand, being able to use them willy-nilly (iff desired) might not be such a bad thing – it could lead to more readable code in the end. Something to think about.

35 DistributionEnsemble

A *distribution ensemble* is a sequence $\{D_n\}_{n \in \mathbb{N}}$ where each D_n is a distribution with finite support on some set Ω .

```
(defn distribution-ensemble ()
  (seq (and (distribution (sub D n) :on Omega)
            (finite-support (sub D n))
            :st (elt n N))))
```

Notes. Not only am I not sure if I parsed that right, I don't know if it is nice to load all that information into the first part of a seq. In general, the idea of a seq needs work.

36 CardinalityOfTheRationals

The set of rational numbers \mathbb{Q} is countable, and therefore its cardinality is \aleph_0 .

```
(defthm rationals-are-countable ()
  (rationals Q)
  (countable Q))
```

Notes. A whole 'nother way to put that would be

```
(defthm rationals-are-countable ()
  (rationals Q)
  (eq (card Q) aleph-null))
```

(Maybe that is what countable translates into.)

37 HamiltonianCycle

let G be a graph. If there's a cycle visiting all vertices exactly once, we say that the cycle is a *hamiltonian cycle*.

```
(defn hamiltonian-cycle (G C)
  (graph G)
  (cycle C G)
  (forall v :in (vertices G) :st (appears-once-in v (vertices C))))
```

Notes. Compare Hamiltonian path! Also, the 'l' at the beginning of the first sentence should be an "L" – submit a correction to the object's owner!

38 RollesTheorem

Rolle's theorem. If f is a continuous function on $[a, b]$, such that $f(a) = f(b) = 0$ and differentiable on (a, b) then there exists a point $c \in (a, b)$ such that $f'(c) = 0$.

```
(defthm rolles-theorem (f a b)
  (function f (closed-interval a b))
  (continuous f)
  (differentiable f (open-interval a b))
  (exists c :in (open-interval a b) :st (eq ((deriv f) c)
                                             0)))
```

Notes. The second argument of continuous defaults to the domain of the function of course. The second argument of function defaults to \mathbb{R} . A variant of Rolle's theorem should be true in the complex case too (right?) but such a theorem would be listed separately. closed-interval will check that its arguments are actually numbers.

39 ExactlyDetermined

An *exactly determined* system of linear equations has precisely as many unknowns as equations and is hence soluble.

```

(defn exactly-determined (X)
  (system-of-linear-equations X)
  (eq (card (unknowns X))
      (card (equations X))))

(defthm every-exactly-determined-system-is-solvable ()
  (exactly-determined X)
  (solvable X))

```

40 Multiset

A *multiset* is a set for which duplicate elements are allowed.
 For example, {1, 1, 3} is a multiset, but not a set.

```

(defn multiset (X)
  (list X))

```

Notes. In fact, this may not be a great translation, but the subtleties of the original are escaping me for now. Being able to translate a vague definition like this requires a lot of expert knowledge. What does it really mean to be a “set for which duplicate elements are allowed”? (Probably a correction should be submitted to the object’s owner.)

41 CrossingLemma

The crossing number of a graph G with n vertices and $m \geq 4n$ edges is

$$\text{cr}(G) \geq \frac{1}{64} \frac{m^3}{n^2}.$$

```

(defthm crossing-lemma (G)
  (graph G)
  (let ((n (card (vertices G)))
        (m (card (edges G))))
    (if (geq m (* 4 n))
        (geq (crossing-number G)
              (* (/ 1 64)
                  (/ (expt m 3)
                     (expt n 2)))))))

```

Notes. The definitions of n and m can be extracted from the definition of G , which is why G is the only argument to this defthm.

42 EulerFermatTheorem

Given $a, n \in \mathbb{Z}$, $a^{\phi(n)} \equiv 1 \pmod{n}$ when $\gcd(a, n) = 1$, where ϕ is the Euler totient function.

```
(defthm euler-fermat-theorem ()
  (integer a n)
  (euler-totient-function phi)
  (if (eq (greatest-common-divisor a n) 1)
      (cong (expt a (phi n))
            1 :mod n)))
```

Notes. Just a simple note on translation: “Given $a, n \in \mathbb{Z}$ ” is translating into the test `(integer a n)` here. (There is nothing particularly noteworthy about this one, I just thought I would make a note of the general pattern.)

43 GradedRing

Let G be an abelian group. A G -graded ring R is a direct sum $R = \bigoplus_{g \in G} R_g$ indexed with the property that $R_g R_h \subseteq R_{gh}$.

```
(defn graded-ring (G R)
  (abelian-group G)
  (ring R)
  (forall g h :in G :st (subset (* (sub R g)
                                   (sub R h))
                               (sub R (* g h))))
  (cartesian-product-1 G R))

(defn cartesian-product-1 (J X)
  (set J)
  (set X)
  (setof (list (sub x j) &c)
         :st (and (elt x X)
                  (elt j J))))
```

Notes. I’m enjoying the cleverness of the `&c` token, but I wonder if it isn’t just a little bit over the top to imagine that a computer could actually interpret the different uses. (It seems a particularly odd to use such a token with `nil` after it.) Again, infinite lists (and potentially uncountably infinite lists, as we have here) need work.

As a separate issue, note that I am really heavily overloading the `*` symbol here. Again, high expectations for the computer are behind this. I use it as a generic multiplication operator whenever I feel like it!

It is interesting to observe that the index set J used in cartesian-product-1 could be uncountable. This is another variant on cartesian product that should probably be in the “standard library”.

44 Monoid

A monoid is a semigroup G which contains an identity element; that is, there exists an element $e \in G$ such that $e \cdot a = a \cdot e = a$ for all $a \in G$.

```
(defn monoid (G)
  (semigroup G)
  (exists e :in G :st (forall a :in G :st (eq (* a e)
                                             (* e a)
                                             a))))
```

Notes. Do we want to give *the identity* a special name, so we could just write something like (has-identity G)?

45 JordansInequality

Jordan’s Inequality states

$$\frac{2}{\pi}x \leq \sin(x) \leq x, \forall x \in [0, \frac{\pi}{2}]$$

```
(defthm jordans-inequality ()
  (forall x :in (closed-interval 0 (/ pi 2))
    :st (leq (* (/ 2 pi) x)
            x)))
```

Notes. Notice that pi here is just treated like a typical symbol; again, we’re expecting the computer to figure out what’s going on. (Also, I fixed a formatting error, should report that to the owner.)

For purposes of understanding this translation, it is well worth noting that in typical mathematical writing, it is about as common for the \forall to come after the statement that it is quantifying as it is for it to come before.

As a minor correction, the author should be using variable sized brackets around 0 and $\pi/2$.

46 DigitalTree

A *digital tree* is a tree for storing a set of strings where nodes are organized by substrings common to two or more strings. Examples of digital trees are digital search trees and tries.

Notes. I just submitted a correction to the author of these “digital trees” entries asking for some clarification!

47 CartesianProduct

For any sets A and B , the *cartesian product* $A \times B$ is the set consisting of all ordered pairs (a, b) where $a \in A$ and $b \in B$.

```
(defn cartesian-product-2 (A B)
  (set J)
  (set X)
  (setof (list a b)
         :st (and (elt a A)
                  (elt b B))))
```

Notes. This definition is kind of like the first cartesian-product. It would be worthwhile to show that these things are more or less equivalent at some point.

48 PrimeRing

A ring R is said to be a *prime ring* if the zero ideal is a prime ideal.

If R is commutative, this is equivalent to being an integral domain.

```
(defn prime-ring (R)
  (ring R)
  (let ((Z (zero-ideal-of R)))
    (prime-ideal Z R)))
```

```
(defthm prime-ring-commutative-is-integral-domain ()
  (commutative R)
  (iff (prime-ring R)
        (integral-domain R)))
```

Notes. See the Introduction for more comments on the definition of prime-ring and issues related to extracting pieces of a definition.

49 CardinalNumber

A cardinal number is an ordinal number S with the property that $S \subset X$ for every ordinal number X which has the same cardinality as S .

```
(defn cardinal-number (S)
  (ordinal-number S)
  (forall X :st (if (and (ordinal-number X)
```

```
(eq (card X)
    (card S)))
(subset S X)))
```

50 AbsoluteConvergenceOfInfiniteProduct

An infinite product $\prod_{n=1}^{\infty}(1+a_n)$ is said to be *absolutely convergent* if $\prod_{n=1}^{\infty}(1+|a_n|)$ converges.

```
(defn absolute-convergence (P)
  (eq (prod n :from 1
            :to infity
            :of (+ 1 (sub a n))) P)
  (converges (prod n :from 1
                  :to infity
                  :of (+ 1 (abs (sub a n))))))
```

Notes. This illustrates yet again how some things in hyperreal are inspired by loop in CL.

51 PascalsRule

Pascal's rule is the binomial identity

$$\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}$$

where $1 \leq k \leq n$ and $\binom{n}{k}$ is the binomial coefficient.

```
(defthm pascals-rule (k n)
  (integer k n)
  (leq 1 k n)
  (eq (+ (binom n k)
         (binom n (- k 1)))
      (binom (+ n 1) k)))
```

Notes. I'm finally putting in arguments for the theorem as instructed by the introduction! I'll get back to fix up the earlier theorems like this later.

This definition illustrates the use of integer with several arguments (all asserted to be integers) – as we've seen before. It also illustrates the use of leq with several arguments to capture the partial ordering of the three numbers 1, k, and n.

52 Bernoulli's Inequality

Let x and r be real numbers. If $r > 1$ and $x > -1$ then

$$(1 + x)^r \geq 1 + xr.$$

The inequality also holds when r is an even integer.

```
(defthm bernoullis-inequality (x r)
  (real x r)
  (or (> 1 r)
      (even r))
  (> -1 x)
  (geq (expt (+ 1 x) r)
        (+ 1 (* x r))))
```

Notes. I think that I've correctly handled the "also holds when" clause, but I'm not completely sure.

53 Traceable

Let G be a graph. If G has a Hamiltonian path, we say that G is *traceable*.

Not every traceable graph is Hamiltonian. As an example consider Petersen's graph.

```
(defn traceable (G)
  (graph G)
  (exists P :st (hamiltonian-path G P)))

(defthm there-exist-traceable-nonhamiltonian-graphs ()
  (exists G :st (and (traceable G)
                     (not (hamiltonian G)))))

(defexample traceable-nonhamiltonian-graph (G)
  (petersens-graph G)
  (traceable G)
  (not (hamiltonian G)))
```

Notes. Examples are another tricky category. On the one hand, maybe they are no different from a theorem; on the other, maybe they are. As with other kinds of supplementary information, we may want to put an example into a slot as part of a bigger "encyclopedia"- or "KM"-style definition for a given term.

54 SumFree

A set A is called *sum-free* if the equation $a_1 + a_2 = a_3$ does not have solutions in elements of A . Equivalently, a set is sum-free if it is disjoint from its 2-fold sumset, i.e., $A \cap 2A = \emptyset$.

```
(defn sum-free (A)
  (set A)
  (forall a1 a2 a3 :in A
    :st (not (eq (+ a1 a2)
                  a3))))

(defn sum-free-1 (A)
  (set A)
  (eq (intersection A (two-fold-sumset A))
      empty-set))
```

Notes. Of course, A has to be such that there is some $+$ defined on it.

55 Chain

Let $B \subseteq A$, where A is ordered by \leq . B is a *chain* in A if any two elements of B are comparable.

That is, B is a *linearly ordered subset* of A .

```
(defn chain (A B)
  (set A)
  (subset B A)
  (ordered-by A leq)
  (forall b1 b2 :in B
    :st (or (< b1 b2)
            (< b2 b1))))
```

Notes. I'm pretty sure that I've captured the intended meaning of "comparable" but I'm not totally sure. (Lack of expert knowledge on my part.)

56 UnderDetermined

An *under determined* system of linear equations has more unknowns than equations. It can be consistent with infinitely many solutions, or have no solution.

```
(defn under-determined (X)
  (system-of-linear-equations X)
  (> (card (unknowns X))
     (card (equations X))))
```

```
(defthm under-determined-system-can-have-many-or-no-solutions ()
  (under-determined X)
  (or (eq (card (solutions X)) infy)
      (eq (card (solutions X)) 0)
      (eq (card (solutions X)) finite)))
```

Notes. Of course, 0 is finite. Note that I've expanded upon the definition that was given.

57 SimplyConnected

A topological space is said to be *simply connected* if it is path connected and the fundamental group of the space is trivial (i.e. the one element group).

```
(defn simply-connected (X)
  (topological-space X)
  (path-connected X)
  (eq (fundamental-group X)
      trivial-group))
```

58 Solvable

A group G is *solvable* if it has a composition series

$$G = G_0 \supset G_1 \supset \cdots \supset G_n = \{1\}$$

where all the quotient groups G_i/G_{i+1} are abelian.

```
(defn solvable-group (G)
  (group G)
  (exists (sub G 0) &c (sub G n)
    :st (and (eq G (sub G 0))
             (supset (sub G 0) (sub G 1))
             &c
             (supset (sub G (- n 1)) (sub G n))
             (eq (sub G n)
                 trivial-group)
             (forall i :from 1
                       :to n
                       :st (abelian (/ (sub G i)
                                       (sub G (+ i 1))))))))))
```

59 OresTheorem

Let G be a simple graph of order $n \geq 3$ such that, for every pair of distinct non adjacent vertices u and v , $\deg(u) + \deg(v) \geq n$. Then G is a Hamiltonian graph.

```
(defthm ores-theorem (G)
  (simple-graph G)
  (let ((n (order G)))
    (geq n 3)
    (forall u v :in G :st (if (neq u v)
                              (geq (+ (deg u)
                                       (deg v))
                                   n))))))
(hamiltonian G))
```

Notes. I'm putting the conclusion `(hamiltonian G)` outside of the `let` form – bearing in mind that everything up until the last element of a `defthm` is a *test*, whereas the last element is a *assertion*. In fact, the two ways of writing this seem to be mathematically equivalent, but it may be more user friendly to keep the scope of n restricted to the part of the statement in which it is actually being used explicitly. Or maybe this is a more confusing way to write. Anyway, as I said, according to the way I'm currently thinking about `defthm`, the two ways are mathematically equivalent.

60 InjectiveFunction

We say that a function $f: X \rightarrow Y$ is *injective* or *one-to-one* if $f(x) = f(y)$ implies $x = y$, or equivalently, whenever $x \neq y$, then $f(x) \neq f(y)$.

```
(defn injective-map (f X Y)
  (map f X Y)
  (forall x1 x2 :in X
    :st (if (eq (f x)
                (f y))
            (eq x y))))

(defn injective-map-1 (f X Y)
  (map f X Y)
  (forall x1 x2 :in X
    :st (if (not (eq x y))
            (not (eq (f x)
                    (f y))))))

(defsyn 'one-to-one 'injective)
```

61 Field

A *field* is a commutative ring F with identity such that:

- $1 \neq 0$
- If $a \in F$, and $a \neq 0$, then there exists $b \in F$ with $a \cdot b = 1$.

```
(defn field (F)
  (commutative-ring-with-identity F)
  (neq (multiplicative-identity f)
      (additive-identity f))
  (forall a :in F :st (if (neq a 0)
                          (exists b :in F :st (eq (* a b)
                                                    1))))))
```

62 ZeroIdeal

In any ring, the set consisting only of the zero element (i.e. the additive identity) is an ideal of the left, right, and two-sided varieties. It is the smallest ideal in any ring.

```
(defthm multi-sided-zero-ideal (R Z)
  (ring R)
  (eq Z (setof (zero-element R)))
  (left-ideal Z)
  (right-ideal Z)
  (two-sided-ideal Z))
```

Notes. I'm not sure off the top of my head how to code up the idea of "smallest" as it is used here. This is a tricky but important idea that will take some more work.

63 RightTriangle

A triangle ABC is right when one of its angles is equal to 90° (and therefore has two perpendicular sides).

```
(defn right-triangle (T)
  (triangle T)
  (exists a :st (and (angle a T)
                     (eq (measure a)
                         (degrees 90)))))
```

Notes. I'm leaving off that bit about "two perpendicular sides" for now.

64 QSkewPolynomialRing

If (σ, δ) is a q -skew derivation on R , then we say that the skew polynomial ring $R[\theta; \sigma, \delta]$ is a q -skew polynomial ring.

```
(defn q-skew-polynomial-ring (R sigma delta q)
  (ring R)
  (skew-derivation q sigma delta R)
  (polynomial-ring (adjoin R theta :st sigma delta)))
```

Notes. In addition to me not really knowing how to translate this stuff (lack of expert knowledge), there is a knowledge engineering issue here. We seem to be defining a *property* of $R[\theta; \sigma, \delta]$ rather than *defining* this object. This is a somewhat subtle distinction that I'm not sure how to resolve right now.

65 SkewDerivation

A (*left*) skew derivation on a ring R is a pair (σ, δ) , where σ is a ring endomorphism of R , and δ is a left σ -derivation on R .

```
(defn skew-derivation (R sigma delta)
  (ring R)
  (ring-endomorphism sigma R)
  (left-derivation sigma delta))
```

Notes. Again the slight difficulty that we might actually be trying to *return* the pair (σ, δ) ?

66 NilpotentIdeal

A left (right) ideal I of a ring R is a *nilpotent ideal* if $I^n = 0$ for some positive integer n . Here I^n denotes a product of ideals – $I \cdot I \cdots I$.

```
(defn nilpotent-ideal (I R)
  (ring R)
  (ideal I)
  (exists n :in Z+ :st (eq (expt I n) 0)))
```

Notes. Should we perhaps make a “stub” to try to capture the meaning of the sentence “Here I^n denotes a product of ideals – $I \cdot I \cdots I$.”? After all, I^n could denote something different, the n th I , for example. It seems to be primarily expert knowledge about the subject area that allows us to guess what I^n is from the first sentence alone. Perhaps “something to the n ” should default to mean $(\text{expt something } n)$, unless we happen to know from context that it is supposed to be translated differently. (I'm not sure whether any of these atypical cases come up in the corpus we're working with, but I think not.)

67 Substring

Given a string $s \in \Sigma^*$, a string t is a *substring* of s if $s = utv$ for some strings $u, v \in \Sigma^*$. For example, lp , al , ha , $alpha$, and λ (the empty string) are all substrings of the string $alpha$.

```
(defn substring (s t)
  (string s t :in (expt Sigma *))
  (exists u v :in (expt Sigma *)
    :st (eq s (concat u t v))))

(defexample example-of-a-substring-1 ()
  (substring "lp" "alpha"))

(defexample example-of-a-substring-2 ()
  (substring "al" "alpha"))

(defexample example-of-a-substring-3 ()
  (substring "ha" "alpha"))

(defexample example-of-a-substring-4 ()
  (substring "alpha" "alpha"))

(defexample example-of-a-substring-5 ()
  (substring "" "alpha"))
```

Notes. We used some knowledge to associate “the empty string” with the actual empty string in our object language. But (so far at least) we also threw away the little clue we were getting from the author that says that the empty string is usually denoted by λ in day-to-day mathematics. It might be best to try to capture this sort of clue.

68 SetOfPrimeIdealsOfACommutativeRingWithIdentity

The set of all prime ideals of a commutative ring R with identity is called the spectrum of the ring and is denoted $\text{Spec}(R)$.

```
(defn spectrum-of-a-commutative-ring-with-identity (R)
  (commutative-ring-with-identity R)
  (setof P :st (prime-ideal P R)))
```

Notes. Not even bothering with the *notation* now. In a more KM-like system, notation should be stored for later reading and writing.

69 SimpleModule

Let R be a ring, and let M be an R -module. We say that M is a *simple* or *irreducible* module if it contains no submodules other than itself and the zero module.

```
(defn simple-module (R M)
  (ring R)
  (module M R)
  (not (exists N :st (and (submodule N M)
                          (neq N M)
                          (neq N 0))))))

(defsyn 'irreducible-module 'simple-module)
```

70 Operator

Synonym of mapping and function. Often used to refer to mappings where the domain and codomain are, in some sense a space of functions.

Examples: differential operator, convolution operator.

```
(defsyn 'operator 'map)
```

Notes. It isn't really a synonym of "function", but whatever. I have no idea how to code the "often used to refer" part of this definition.

71 FractionField

Given an integral domain R , the *fraction field* of R is the localization $S^{-1}R$ of R with respect to the multiplicative set $S = R \setminus \{0\}$. It is always a field.

```
(defn fraction-field (R F)
  (integral-domain R)
  (let ((S (setminus R (setof 0))))
    (eq F (localization (expt S -1) R))))

(defthm fraction-field-is-a-field (R F)
  (if (fraction-field R F)
      (field F)))
```

Notes. We haven't come up with anything for the words "the multiplicative set". See comments on "stub" entries elsewhere in these notes.

72 CosinesLaw

Let a, b, c be the sides of a triangle, and let A the angle opposite to a . Then

$$a^2 = b^2 + c^2 - 2bc \cos A.$$

```
(defthm cosines-law (a b c A)
  (if (and (exists T :st (and (triangle T)
                              (eq '(a b c)
                                  (sides T))))
        (eq (angle-opposite-side a) A))
      (eq (expt a 2)
          (- (+ (expt b 2)
                (expt c 2))
             (* 2 b c (cos A))))))
```

Notes. I guess it is most appropriate to treat this as a theorem and not as a definition (since after all it must be proved).

73 MonotonicallyDecreasing

A sequence (s_n) is *monotonically decreasing* if

$$s_m < s_n \quad \forall m > n$$

Compare this to monotonically nonincreasing.

```
(defn monotonically-decreasing (S)
  (seq S)
  (forall (sub S m) (sub S n) :in S :st (if (> m n)
                                             (< (sub S m)
                                                (sub S n))))))
```

74 Irrational

An *irrational number* is a real number which cannot be represented as a ratio of two integers. That is, if x is irrational, then

$$x \neq \frac{a}{b}$$

with $a, b \in \mathbb{Z}$.

```
(defn irrational-number (x)
  (real x)
  (forall a b :in Z :st (neq x (/ a b))))
```

75 Domain3

Let R be a binary relation. Then the set of all x such that xRy is called the *domain* of R . That is, the domain of R is the set of all first coordinates of the ordered pairs in R .

```
(defn domain-of-binary-relation (R)
  (binary-relation R)
  (setof x :st (exists y :st (R x y))))
```

Notes. We need some knowledge about “binary relations” to be able to see that xRy is a relationship between x and y , of the kind that is described by a binary predicate in our object language. Without this knowledge, we might have translated xRy as $(* x R y)$.

76 PID

A *principal ideal domain* D is an integral domain where every ideal is a principal ideal.

In a PID, an ideal (p) is maximal if and only if p is irreducible (and prime since any PID is also a UFD).

```
(defn principal-ideal-domain (D)
  (integral-domain D)
  (forall I :st (if (ideal I D)
                    (principal-ideal I))))
```

```
(defthm ideal-maximal-iff-reducible (D p)
  (principal-ideal-domain D)
  (elt p D)
  (iff (irreducible p)
        (maximal (ideal-generated-by p))))
```

```
(defthm pid-is-ufd (D)
  (if (principal-ideal-domain D)
      (ufd D)))
```

```
(defthm ideal-prime-if-maximal (D p)
  (principal-ideal-domain D)
  (elt p D)
  (let ((I (ideal-generated-by p)))
    (if (maximal I)
        (prime I))))
```

```
(defsyn 'principal-ideal-domain 'pid)
```

Notes. There is a surprising amount of information crammed into these few sentences. The bit about the (ideal-generated-by p) comes from “(p)” – I think I translated this correctly, but it takes knowledge!

77 LinearlyOrdered

An ordering \leq (or $<$) of A is called *linear* or *total* if any two elements of A are comparable. The pair (A, \leq) is then called a *linearly ordered set*.

```
(defn linear-ordering (S A)
  (set S)
  (ordering S A)
  (forall a b :in A :st (defined-p (S a b)))
  'A)

(defn linearly-ordered-set (S A)
  (linear-ordering S A)
  '(S A))
```

Notes. This section nicely illustrates the demand for having different *return types* for different definitions. This should be addressed everywhere, and a good default behavior should be created. This relates to the problem of accessing different “parts” of a definition (for example, the group operation that makes up part of a group).

78 SuperfluousSubmodule

Let X be a submodule of a module Y . We say that X is a *superfluous submodule* of Y if whenever A is a submodule of Y such that $A + X = Y$, then $A = Y$.

```
(defn superfluous-submodule (Y X)
  (module Y)
  (submodule X Y)
  (forall A :st (if (and (submodule A Y)
                          (eq (+ A X)
                               Y))
                    (eq A Y))))
```

79 LipschitzFunction

A *Lipschitz function* is a function $f: \mathbf{R} \rightarrow \mathbf{C}$ for which there exists an $M \in \mathbf{R}$ such that for all $x, y \in \mathbf{R}$

$$|f(x) - f(y)| \leq M|x - y|.$$

```
(defn lipschitz-function (f)
  (function f R C)
  (exists M :in R :st (forall x y :in R :st (leq (abs (- (f x)
                                                         (f y)))
          (* M (abs (- x
                    y))))))))
```

80 Irreducible

Let D be an integral domain, and let r be a nonzero element of D . We say that r is *irreducible* in D if for any factorization $r = ab$ in D we must have that a or b is a unit.

```
(defn irreducible (D r)
  (integral-domain D)
  (non-zero-elt r D)
  (forall a b :in D :st (if (eq r (* a b))
                            (or (unit a)
                                (unit b))))))
```

81 MonotonicallyIncreasing

A sequence (s_n) is called *monotonically increasing* if

$$s_m > s_n \forall m > n$$

Compare this to monotonically nondecreasing.

```
(defn monotonically-increasing (S)
  (seq S)
  (forall (sub S m) (sub S n) :in S :st (if (> m n)
                                            (> (sub S m)
                                              (sub S n))))))
```

Notes. I haven't coded up the "compare..." stuff here or elsewhere. Would probably be good to have a rule to handle this stuff.

82 HeronsFormula

The area of a triangle with side lengths a, b, c is

$$\Delta = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = \frac{a+b+c}{2}$ (the semiperimeter).

```

(defthm herons-formula (a b c)
  (if (exists T :st (and (triangle T)
                        (eq '(a b c)
                            (sides T))))
      (let ((s (semiperimeter T)))
        (eq (area T)
            (sqrt (* s
                    (- s a)
                    (- s b)
                    (- s c)))))))

(defn semiperimeter (T)
  (triangle T)
  (/ (+ (side 1 T)
        (side 2 T)
        (side 3 T))
     2))

```

83 Homeomorphism

A *homeomorphism* f of topological spaces is a continuous, bijective map such that f^{-1} is also continuous. We also say that two spaces are *homeomorphic* if such a map exists.

```

(defn homeomorphism (f X Y)
  (topological-space X Y)
  (continuous f X Y)
  (bijection f X Y)
  (continuous (inv f) (img f X) X))

(defn homeomorphic (X Y)
  (exists f :st (and (map f X Y)
                    (homeomorphism f X Y))))

```

84 FlatResolution

Let M be a module. A *flat resolution* of M is an exact sequence of the form

$$\cdots \rightarrow F_n \rightarrow F_{n-1} \rightarrow \cdots \rightarrow F_1 \rightarrow F_0 \rightarrow M \rightarrow 0$$

where each F_n is a flat module.

```

(defn flat-resolution (M)
  (module M)
  (let (((sub F 0) (sub F 1) &c))
    (forall n :in N :st (flat-module (sub F n)))
    (exact-seq &c (sub F n) &c (sub F 0) M 0)))

```

Notes. The difference between *the* such-and-such and *a* such-and-such is important, but I probably haven't been paying enough attention to the difference so far. I don't remember if I have elsewhere dealt with the problem of instantiating a sequence of things; the way it is done here might not be particularly nice. Also the notion of an exact sequence used here is probably not quite what is wanted, but developing the correct notion will take some work.

It seems like it would be better to take $F_0 \dots F_n$ as *arguments* to this defn, since they don't show up as "if there exist $F_0 \dots F_n$ " or anything like that – they seem to be just as important as M . But this poses some notational challenges. Maybe this is just a LISP knowledge gap; if I could refer back to the ARG list, I could just say something like

```
(defn flat-resolution (M FO F1 &c)
  (module M)
  (forall n :in N :st (flat-module (ARG n)))
  (exact-seq &c F1 FO M O))
```

Probably you can do *something* like this in LISP but I'm not sure how.

85 FreeResolution

Let M be a module. A *free resolution* of M is an exact sequence of the form

$$\dots \rightarrow F_n \rightarrow F_{n-1} \rightarrow \dots \rightarrow F_1 \rightarrow F_0 \rightarrow M \rightarrow 0$$

where each F_n is a free module.

```
(defn flat-resolution (M)
  (module M)
  (let ((sub F 1) &c))
  (forall n :in N :st (free-module (sub F n)))
  (exact-seq (sub F n) &c (sub F 0) M O)))
```

Notes. Don't be confused - this isn't the same as the last definition!

86 SylowsFirstTheorem

If G is a finite group and p is a prime such that p^k divides $|G|$, then there is a subgroup H of G such that $|H| = p^k$.

This is the first part of several results usually called the Sylow theorems.

```
(defthm sylows-first-theorem (G p k)
  (finite-group G)
  (prime p)
  (divides (expt p k) (order G))
  (exists H :st (and (subgroup H G)
                     (eq (order H)
                         (expt p k))))))
```

Notes. I don't know how to code the "this is the first part..." bit.

87 ApolloniusTheorem

Let a, b, c the sides of a triangle and m the length of the median to the side with length a . Then $b^2 + c^2 = 2m^2 + \frac{a^2}{2}$.

```
(defthm apollonius-theorem (T)
  (let* ((a (side 1 T))
         (b (side 2 T))
         (c (side 3 T))
         (m (median-to a)))
    (eq (+ (expt b 2)
           (expt c 2)
           (+ (* 2 (expt m 2))
              (/ 2 (expt a 2))))))
```

Notes. Probably it is better to feed in the triangle T than a, b and c . Just think about how the theorem would be used. (Previous things with triangles may need to be adjusted accordingly.) I have no idea what "median to the side with length a " means.

88 Region

A region is a connected domain.

Since every domain of \mathbb{C} can be seen as the union of countably many components and each component is a region, we have that regions play a major role in complex analysis.

```
(defn region (R)
  (domain R)
  (connected R))

(defthm every-complex-domain-is-union-of-countably-many-components ()
  (forall D :st (if (and (domain D)
                        (subset D C))
                    (exists (sub A 1) (sub A 2) &c
                          :st (forall n :in N
                                :st (region (sub A n)))))))
```

Notes. Here the $\&c$ takes care of the hypothesis of countability. I have no idea how to code the rather confusing statement "we have that regions play a major role in complex analysis." This statement appears here without enough context to make it make sense.

89 Semigroup

A *semigroup* G is a set together with a binary operation $\cdot : G \times G \rightarrow G$ which satisfies the associative property: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in G$.

```
(defn semigroup (G *)
  (set G)
  (operation * G)
  (associative * G))

(defn associative (* X)
  (and
    (operation * X)
    (forall a b c :in X :st (eq (* (* a b)
                                   c)
                                (* a
                                   (* b c))))))
```

90 UniformModule

A module M is said to be *uniform* if any two nonzero submodules of M must have a nonzero intersection. This is equivalent to saying that any nonzero submodule is an essential submodule.

```
(defn uniform-module (M)
  (module M)
  (forall N1 N2 :st (if (and (submodule N1 M)
                              (submodule N2 M))
                        (neq (intersection N1 N2) 0))))

(defthm uniform-module-iff-all-nonzero-submodules-essential (M)
  (iff (uniform-module M)
        (forall N :st (if (submodule N M)
                          (essential-submodule N M)))))
```

91 MonotonicallyNondecreasing

A sequence (s_n) is called *monotonically nondecreasing* if

$$s_m \geq s_n \quad \forall m > n$$

Compare this to monotonically increasing.

```
(defn monotonically-nondecreasing (S)
  (seq S)
  (forall (sub S m) (sub S n) :in S :st (if (> m n)
```

```
(geq (sub S m)
      (sub S n))))
```

92 MorleysTheorem

The points of intersections of the adjacent trisectors in any triangle, are the vertices of an equilateral triangle.

```
(defthm morleys-theorem (T)
  (equalateral-triangle
   (intersection
    (setof (adjacent-trisectors T))))))
```

Notes. This was coded without having any clue what “adjacent trisectors” are. I hope these are defined in PlanetMath!

93 SquareOfANumber

The square of a number x is the number obtained multiplying x by itself. It's denoted as x^2 .

Some examples:

$$\begin{aligned}5^2 &= 25 \\ \left(\frac{1}{3}\right)^2 &= \frac{1}{9} \\ 0^2 &= 0 \\ .5^2 &= .25\end{aligned}$$

```
(defn square (x)
  (number x)
  (* x x))

(defexample example-of-a-square-1 ()
  (eq (square 5) 25))

(defexample example-of-a-square-2 ()
  (eq (square (/ 1 3)) (/ 1 9)))

(defexample example-of-a-square-2 ()
  (eq (square 0) 0))

(defexample example-of-a-square-1 ()
  (eq (square .5) .25))
```

94 SecondCountable

A topological space is said to be *second countable* if it has a countable basis. It can be shown that a second countable space is both Lindelöf and separable, although the converses fail.

```
(defn second-countable (T)
  (topological-space T)
  (exists B :st (and (basis B T)
                    (countable B))))

(defthm every-second-countable-space-is-lindelöf-and-separable (T)
  (if (second-countable T)
      (and (lindelöf T)
           (separable T))))

(defthm not-every-lindelöf-and-separable-space-is-second-countable-space ()
  (exists T :st (and (lindelöf T)
                    (separable T)
                    (not (second-countable T)))))
```

Notes. It might be nice to be able to capture the truth of the statement and the failure of its converse inside one top-level LISP form.

95 GradedModule

If $R = R_0 \oplus R_1 \oplus \dots$ is a graded ring, then a graded module over R is a module M of the form $M = \bigoplus_{i=-\infty}^{\infty} M_i$ and satisfies $R_i M_j \subseteq M_{i+j}$ for all i, j .

```
(defn graded-module (R)
  (let (RO R1 &c)
    (and (graded-ring R)
         (eq R (oplus RO R1 &c))))
  (let (MO M1 &c)
    (and (module M)
         (eq M (oplus i :from -infty :to infty :of (sub M i)))
         (forall i j :st (subset (* (sub R i) (sub M j)))))))
```

Notes. The definition as it was originally written is potentially somewhat inscrutable. What exactly is the set that i ranges over? I'm kind of leaving this out of my code... but no worse than the original.

96 ParallelogramLaw

Let $ABCD$ be a parallelogram with side lengths u, v and whose diagonals have lengths d_1 and d_2 then

$$2u^2 + 2v^2 = d_1^2 + d_2^2.$$

```
(defthm parallelogram-law (P)
  (parallelogram P)
  (elt (lengths (sides P)) (setof u v))
  (elt (lengths (diagonals P)) (setof d1 d2))
  (eq (+ (* 2 (expt u 2))
        (* 2 (expt v 2)))
      (+ (expt d1 2)
         (expt d2 2))))
```

Notes. This version of the translation violates my “rule” that un-identified symbols shouldn’t be introduced willy-nilly into the code. On the other hand, maybe this way of writing is more natural. Should be considered.

97 TuringComputable

A function is *Turing computable* if the function’s value can be computed with a Turing machine.

For example, all primitive recursive functions are Turing computable.

```
(defn turing-computable (f)
  (exists T :st (and (turing-machine T)
                    (computes-value-of T f))))

(defthm primitive-recursive-functions-are-turing-computable ()
  (if (primitive-recursive-functions f)
      (turing-computable f)))
```

Notes. This definition is rather bad – it never says what it means for a function’s value to be computed with a Turing machine. Maybe that information is provided in the PM encyclopedia – but then I question the usefulness of this definition.

98 AlgebraicNumber

A number $\alpha \in \mathbb{C}$ is called an algebraic number if there exists a polynomial $f(x) = a_n x^n + \dots + a_0$ such that a_0, \dots, a_n , not all zero, are in \mathbb{Q} and $f(\alpha) = 0$.

```
(defn algebraic-number (alpha)
  (elt alpha C)
  (exists f :st (and (polynomial f)
```

```

(forall x :in C
  :st (eq (f x)
          (+ (* (sub a n) (expt x n))
             &c
             (* (sub a 0) (expt x 0))))))
(exists i :in n
  :to 0
  :st (neq (sub a i)
          0))
(forall i :in n
  :to 0
  :st (elt (sub a i) Q))
(eq (f alpha) 0))))

```

Notes. Important! It doesn't really seem necessary to say `(polynomial f)` when we are providing the expansion of the polynomial. On the other hand, probably we don't need to give the expansion if we say that f is a polynomial. Note that the variables `(sub a n)` haven't really been defined anywhere. This should be rectified – maybe `(polynomial f)` would introduce some coefficients into the namespace automatically, that could then be referenced by `(coefficient i x)` for example. This leads to a definition that's probably quite a bit better:

```

(defn algebraic-number (alpha)
  (elt alpha C)
  (exists f :st (and (polynomial f)
                     (exists i :in n
                       :to 0
                       :st (neq (coefficient i f) 0))
                     (forall i :in n
                       :to 0
                       :st (elt (coefficient i f) Q))
                     (eq (f alpha) 0))))

```

This one should be taken to be the official definition. (Hopefully I'll get around to revising this section further sometime soon.)

99 DirectImage

Let $f : A \rightarrow B$ be a function, and let $U \subset A$ be a subset. The *direct image* of U is the set $f(U) \subset B$ consisting of all elements of B which equal $f(u)$ for some $u \in U$.

```

(defn direct-image (f A B U)
  (map A B)
  (subset U A)
  (f U))

```

Notes. Well, overloading `f` to apply it to `U` might not be such a good idea. There's a little more helpful redundancy in this definition that should be coded up at some point.

100 InductiveSet

An *inductive set* is a set X with the property that, for every $x \in X$, the successor x' of x is also an element of X .

One major example of an inductive set is the set of natural numbers \mathbb{N} .

```
(defn inductive-set (X)
  (set X)
  (forall x :in X :st (elt (successor x) X)))

(defexample example-of-inductive-set ()
  natural-numbers)
```

Notes. This makes me remember that “examples” are actually theorems (and so need proofs).

Appendix A: Language Specification

<code>(integer &rest INTEGERS)</code>	Each of the INTEGERS is in fact an integer.
<code>(set &rest SETS)</code>	Each of the SETS is in fact a set.
<code>(subset &rest SETS)</code>	Each of the SETS is contained in the next set. For example, <code>(subset A B C)</code> means $A \subset B \subset C$. Error if there are less than two SETS.
<code>(intersection &rest SETS)</code>	Find the intersection of the given SETS. If there is only one set given, find the intersection of its elements.
<code>(union &rest SETS)</code>	Find the union of the given SETS. If there is only one set given, find the union of its elements.
<code>(supset &rest SETS)</code>	Each of the SETS is contained in the previous set. For example, <code>(supset A B C)</code> means $A \supset B \supset C$. Error if there are less than two SETS.
<code>(cong A B :mod C)</code>	A , B , and C are integers; A is congruent to B modulo C . (How is this idea expressed in LISP? We might want to do something more like that.)
<code>(iff THIS THAT)</code>	Check or assert that THIS is true if and only if THAT is true.

(defn NAME ARGLIST [DOCSTRING] BODY...)

This is kind of like a defun in LISP. But not quite (the definition needs some tuning).

(defthm NAME ARGLIST [DOCSTRING] BODY...)

This is kind of like a defun in LISP. But not quite (the definition needs some tuning).

(exists OBJECTS [:in SPACE] [:st PROP])

create new OBJECTS, elements of SPACE, with property PROP. If there are two or three arguments, the :in and :st keywords are optional. In the case of two arguments, (exists foo bar) means that there is an object foo with property bar (the space in which foo lives is found from context). The :in SPACE instruction is semantically equivalent to a PROP instruction that includes (elt OBJECT SPACE). One of SPACE or PROP must be supplied. In the case of an index ranging over a countable set, :from LB :to UB can be substituted for :in SPACE.

(forall OBJECTS [:in SPACE] [:st] PROP)

Assert PROP for every object among OBJECTS, optionally restricted to be elements of SPACE. (If SPACE is not supplied, the space in which the objects lives is found from context.) If there are two or three arguments, the :in and :st keywords are optional. In the case of two arguments, (forall foo bar) means that every foo has the property bar (the space in which foo lives is found from context). The :in SPACE instruction is semantically equivalent to to a PROP instruction that includes (if (elt OBJECT SPACE) ...). In the case of an index ranging over a countable set, :from LB :to UB can be substituted for :in SPACE.

&c

Repetition token. Shorthand to be used inside a list that is indexed by a natural numbers. It can stand for a finite or infinite number of omitted terms.

(setof OBJECTS [:in SPACE] [:st] PROP)

Extract the collection of OBJECTS in SPACE with property PROP. If SPACE is not supplied, the space in which the objects lives is found from context. In the case of an index ranging over a countable set, :from LB :to UB can be substituted for :in SPACE.

(prod INDEX :from LB :to UB :of DEP)

The product of DEP as INDEX ranges from LB to UB. If the space in which the index lives is not countable, :in SPACE can be substituted for :from LB :to UB.

(sub INDEX :from LB :to UB :of DEP)

The sum of DEP as INDEX ranges from LB to UB. If the space in which the index lives is not countable, :in SPACE can be substituted for :from LB :to UB.

(seq OBJECTS :st COND)

OBJECTS are indexed according to COND. (For example, COND might be (elt n N) if the sequence is countable; there would then be an object in the list OBJECTS for each element of \mathbb{N} .)

empty-set

EMPTY-SET is a constant, always equal to the empty set.

(exact-seq [:of] GROUPS [:with-maps MAPS])

GROUPS is a finite or infinite sequence of groups. The operations that map between the groups need not be specified explicitly; if they are specified as MAPS, the :with-maps keyword must appear and the indexing of MAPS should be the same as the indexing of GROUPS (the first map will apply to the first group, etc.).

Notes. You can use

(forall (oplus x y) :in (oplus X Y) :st (prop x y))

as shorthand for

(forall x :in X :st (forall y :in Y :st (prop x y)))

(though the observant reader will see that these expressions in fact have the same length).

I think that the repetition token should be seen as particularly controversial at this point (I'm not sure how to deal best with omitted variables however).