SpecL/FMSL/RSL Implementation Ideas

Nil, [], and '' (26feb09)

To be formal, the type of the empty-list value "`[]`" is `?T*`. Similarly, the type of the valueless function value "`''`" is `op(?T)->(?T)`. The latter definition may be unnessary, overly complicated, and stupid. If it really is, then we'll just say that `''` is a synonym for nil for any opaque type or valueless function. If we need to make this work formally, or even reflexively in SpecL, then it would appear that we need some way to define "opaque" type in the SpecL type algebra.

At this point, I'd say that the easy way out is to represent `''` formally as a special case overloading of a bit of syntactic sugar, that's easy enough to define in the formal semantics, but does not need to be defineable directly in SpecL, e.g., as something like this would-be definition:

```
obj '' = (op(?T)->(?T)) or ?T|is_opaque(?T);
```

Outputless Ops (26feb09)

They are offcially "valueless" operations, that always return nil, and have no type. This begs the question as to whether the keyword "`nil`" should be usable as a type ident. The bottomline is that I say NO. Rataionnale follows.

While `NilType` is in fact an internal interpreter type, it make things too loose to allow it to be used in a source specification. Here's why. Having any data ident be declared NilType is useless, because there's now way to meaningfully allocate storage for it.

I was initially tempted to say that an op with not explicit return type was an "opaque" type, but that seems not to be sensible. What we can say is that the sym lit `''` is the only existing value for a valueless function. We can have `''` be a synonym for nil, but for operation types, in the same way that `[]` is a synonym for nil for list types. There seems to be some degenerate consistency here, though it may be more complicated than it's worth. See the preceding item in these notes for further discussion.

Setting Output Parms by Name (25feb09)

As if we need yet another unsettled "idea", here's one. It may be worthwhile to consider an exception to the would-be rule that `set` can only be used on declared `vars`, that exception being the use of `set` on output parms. So, motherfucker, consider it.

One More Time, and the LAST Time, for "Attribute" versus "Property" (20feb09)

At this point, I think it's largely futile to worry about having any substantial agreement with SpecL versus UML terminology. Where it doesn't significantly comprimise SpecL syntax and/or semantics, then SpecL/UML agreement is fine. Otherwise, F it.

A particular case in point is the long-standing anxiety over the UML use of the term "attribute" compared to the SpecL use. Having read Wikipedia, of all places (the article on Entity-attribute-value model), I've become fully comfortable with the SpecL sense of the term being worth keeping, and hence we will NOT rename SpecL "attributes" to "properties".

We'll have an appropriate section in the ref man that discusses the issue thoroughly, perhaps even with a small dose of whimsy. The idea is this: "Look at the UML, understand it in terms of the underlying SpecL, and don't worry about the differences in UML versus SpecL terminology. A UML diagram is-a UML diagram. By any other name, it would smell just as bad."

Most Recent Thoughts on Grammar Rules, Symlits, and Multi-Paradignmness (20feb09)

Keep grammar rules, but don't make them synomous with obj defs. Keep symlits, as the way to represent something that has mnemonic name, but is not modelwise considered to be a string. (We could forget the second of these if we redo the latest term-factor example with double-quoted strings for terminals instead of symlits, but having looked at it yet again, symlits are better, if not significantly so, as terminal symbols in a grammar, as well as other places that we've cited often enough, including

a. as an enumeration literal, in lieu of a string. Do have a section on this in the ref man, that does a side-by-side comparison of the two representations of, say, a DaysOfTheWeek enumeration, as a string union versus a symlit union. Explain, explain, explain.

A FIRM DECISION: We should in fact have separeate syntax for a new type of entity called a "rule". It can look like a conventional grammar rule, so that the latest and greatest version of the term-factor attribute still works. I thought for a bit about having a new '`rule`' keyword, but that would make grammar rules look stupid. Another good reason to separate grammar rules from obj defs is that they're really not the same entities, despite the areas of similarity. In particular, embedded actions really don't make sense for obj defs, and inheritance doesn't seem to make much if any sense for grammar rules.

There is the kitechen-sink-esque nature of having grammar rules as a separate language feature, but I think it can be ((reasonably) well) rationalized on the grounds that SpecL is, after all, billed as a "multi-paradignm" specification language. Two whit, the semantic paradigms that are supported are:

a. model-based/predicative
b. algebraic/equational
c. functional/operational
d. imperative/operational
e. dataflow
f. entity/relationship (not sure this is separate from one or more of the above)
g. attribute grammar
h. possibly state-machine

Text for Ref Man Discussion of Vars, Etc. (16feb09)

"... There is a simple rule to remember about specifications that use variables -- if you want your specification to be purely functional, don't have any variable declartions."

"So, what is the differece between a let expression with and wihtout an explicit type declaration? It's subtle, having to do with inheritance. Also worth noting is the issue of testing value universes. *(These both need to be explained thoroughly.)*"

Overloading WRT Type Vars, and Nixing Coarity Overloading (15feb09)

For the purposes of overload resolution, a type var anywhere in a signature subsumes overloading at the type var position. E.g., the following results in a function redef error:

```
op O1(x:?X, y:integer, z:string);
op O1(y:?Y, y:integer, z:string);
```

as does

```
op O1(x:?X, y:integer, z:string);
op O1(x:integer, y:integer, z:string);
```

For even more symplicity, it may well be worthwhile to say that a type var *anywhere* in a signature subsumes overloading at all. E.g., the following also results in a redef error, or more clearly, an error described as redef of a polymorphic function:

```
op O1(x:?X, y:integer, z:string);
op O1(...);   // For *any* zero or more parameters in "..."
```

And for even further simplicity, we will nix coarity overloads. (Motherfucker -- this decision was made firmly 8.5 years ago, in the 21sep00 entry! This is one shocking and scary motherfucker of an observation, motherfucker. Time to get things the fuck done, once and for motherfucking all. Did I say Motherfucker?!?)

No More Grammar Rule Cuteness, and Other Syntactic Musings (13feb09)

Given how ill-used the keywordless, space-delimited obj definitions are likely to be, and how not-really-that-messy a file like term-factor-does-work.rsl looks, I think it's time to simplify the fuck out of things by getting rid of the syntax that allows objects to be declared without an "obj" keyword, and with space-delimited componentn exprs.

Once this is done, parens will be a better means to declare parameterized types than angle brackets, since the latter most likely causes syntactic clash, if no visual confusion with '>' as the inheritance shortcut.

And another (?final for now?) syntactic observation is that we may well want to allow all type expression ops in an op signature, to allow the ready definition of functions like the ML for loops. We classes-archives/530/examples/ml/for.ml and the rudimentary attempt to duplicate this in newer-inputs/for.fmsl.

Let Semantics, Redux (14feb09)

OK, how bout we screw ML, and have genuine single-assessment lets, meaning in particular that you can't reassign a let var the way you can an ML value. This will make things nicer pretty much all around, since more than one appearance of let var on the LHS of a let is a redeclaration error. Lets can stil, of course, appeare on the RHS of another let, as long as they're declared lexcically before their RHS use. This makes lets like let* in Lisp, in that order is in fact relevant.

Testing Ideas (11feb09)

Provide different levels of axiom validation:

a. after every operation invocation (default)
b. after every expression evaluation, i.e., expressions in an actual parameter list or expression sequence
c. after every boolean clausew evaulation, i.e., after every boolean AND, OR, or NOT operator eval
d. after every operator evaluation, i.e., after the evaluation of any and all sub-expression operators

Semantics of let, val, and expression sequences (4feb09)

OK, here are conclusions related to the 3feb09 discussion of this topic:

a. Let's can only appear in an explicit expression sequence, which means they cannot be used raw, immediately after a top-level prompt. An expr list is legal at the top-level prompt, but it creates its own local scope, such that any lets it contains go away at the close of that scope, i.e., at the end of that expression seqence's execution.
b. We will change the syntax of the anachronistic "desig := expr" to "set desig = expr". I think this is a much more sensible syntax, and we can explain clearly in a ref man section entitled "The Difference Between `let` and `set`". To whit:
   i. only names, not designators can appear on the LHS of the let '=';
   ii. let's are "single assignment" local variables, blah, blah, blah

iii. Values, declared with '`val`', are strictly one-time declared global constants. They cannot be redeclared, excepted within module boundaries. They are not executable expressions in any way, so they cannot appear in expr lists, as in ML. Value exprs must be evaluatable before runtime, and hence cannot contain variables, or be defined circularly.
iv. Variables, declared with '`var`', are strictly one-time declared globals. They cannot be redeclared, excepted within module boundaries. They are not executable expressions in any way, so they cannot appear in expr lists. Var

intialization exprs must be evaluatable before runtime, and hence cannot contain unitialized variables, or be defined circularly.

v. Expression sequences define their own open scope, with `let` being the sole form of declaration within the scope. Within expr sequences, let and set identifiers are completely separate. In particular, the `let` declaration of a local identifier *X* hides any let var *X* defined in an outer scope, and any variable named *X*. Hence, a `let` var has a different class than a global var, and they are not interchangeable. A clash between let and set vars can only happen within an expression sequence, and only in the following circumstance:

> A let var is declared, and subsequently appears on the LHS of a set; the typecheck error message in this case is "cannot use set on a local variable declared with let".

Note that the converse of this will not cause a clash, since a global variable name appearing on the LHS of a let will redeclare that var name locally, and hence hide the global.

Semantics of let, val, and expression sequences (3feb09)

Doing work on the interp has led to reconsideration of these semantics, and their implementation. Here are some initial notes, with more to follow, perhaps on a different date.

For the implementation of value execution, it can be done as last pass in the type checker, after all other checking has been done. Alternatively, it could be done as first pass in the interpreter. Whether values are constants, or "lettable" is discussed further below. I'm strong inclined to keep them constants.

To avoid confusion between '`let`' and '`:=`', we could say that let's not legal at the top-level of the interp, which can be done however it needs to happen. Probably only a single expr should follow the '>' "prompt", instead of an expr list. This shoud take care of lets not being allowed at the top level. Explicit paren-enclosed expr lists will still be allowed at the top level, but the lets, if any, in these contexts willl be local only to those anonymous scopes, and therefore not survive out to the top level. What his means, also, is that only global var decls and assignment can be used at the top level to store intermediate sandbox values. This may well be fine.

I've considered making the semantics of FMSL values like those of ML, however the problem with this is that it allows values to be redeclared, which is not really what we want in fmsl. For example, if FMSL vals worked like ML, we could do the following:

```
val x = 10;
op f() = x;
> x;
> f();
val x = x + 1;
> x;
> f();
```

outputs

```
10
10
11
10
```

But this seems (has always seemed) rather disengenuous, since x is really acting more like a variable than a value in this case. The non-variable behavior here is that the use f retains the historical value of x, rather than values of x made subsequent to f's declaration. Here's what things could look like in FMSL to get the same semantics:

```
let x = 10;
op f() = x;
> x;
> f();
let x = x + 1;
> x;
> f();
```

The deal here is that let is a functional single-assignment thing, that temporally redeclares a variable, such that any previous uses refer to a different value than a subsequent redclaration. This seems a bit better than ML, though it does still have the "temporal surprise" issue.

Let's try to rationalize things here, in terms of values, variables, and lets. First, a `value` declaration is a constant through the lifetime of a specification. It is always an error to re-declare a value identier. Also, a value declaration is a top-level entity declaration, NOT an expression. This means a value declaration cannot appear in an expression sequence, in particular in a function body. The value identifier itself is an expression value, and can appear in any r-value expression context.

A `variable` is like a `value`, declarationwise. I.e., a `variable` declaration is top-level only, and cannot be re-declared. It models a piece of data that is shared by all operations a functions. A variable identifier is an expression value, and can appear in any r-value or l-value context.

A `let` is both a declaration and an expression. It (re)declares an identifier *in the current scope*, and may appear in any expression context. Specifically, a let of the form

```
let name = expr
```

declares `iname` in the current scope, if it is not already declared. It then binds the value of the `expr` to `name`. A let name can appear in any subsequent r-value context, where it is evaluated by immediate macro expansion. This means, in particular, when a let variable appears as an r-value within the body of an operation declaration, the *evaluated* value of the let varible is placed within the function body, *NOT* the unevaluated let variable name. The most important consequence of this is that if and when the operation is subsequently invoked, it (effectively) uses the most recent bound value of the let name, not any subsequent value that the let variable may have been re-bound to. This behavior is akin to the behavior of `val` in ML.

A let expression, as with any other expression, CANNOT appear in a top-level declaration context, although it can be executed in the top-level execution scope. It's the latter case that gets a little funky, surprisewise. E.g.,

```
> let x = 10;
var x:integer;
ERROR: you cannot redeclare the sweet and inoocent let variable "x"
       to ba a nasty and evil global variable

> x := 10;
ERROR: you cannot change the sweet and innocent let variable "x"
       with the nasty and evil assignment operator ":="

var y: integer;
> let y = 10;
ERROR: you cannot change the nasty and evil global variable "y"
       with the sweet and innocent let operator
```

I'm inclined to use these as actual error message texts, so that the whimsy might help diminish the nastiness of the surprise.

Another possibility is to *syntactically* restrict let exprs to be at the beginning of an expression list. This would be consistent with Lisp and ML, and make things much simpler with regard to mixing and matching global vars and lets.

The General Idea of "Functionalizing" or "Purifying" a Spec (14jan09)

It may well have already been done/thought-of in the literature. Anyway, the idea is to find uses of global vars and term them into explicit function parms. I think it can be done statically, with a transitive closure on the calling chain of each fucntion using one or more globals.

A quick look at the Wikipedia article on parallel computing is a bit depressing in this regard, given that there appears to be a lot of shit I don't know about, even as early as Lamport's 1979 shit of the sequential equivalence of parallel program. Anyway, the idea of "purifying", or at least having the type checker check for pureness may be worthwhile, if not ground breaking.

Syntax of Expression Sequence (27dec07)

To avoid CJ culture shock, consider using/allowing curly braces for expression sequences. This would overload them with their use as tuple constructor, but this is probably OK. One potential ramification of this is the assignment of function values to function-valued variables, but this might just well workout fine. This brings up the issue of free vars in a function body, which presumably we should deal with like Lisp, but have the type cheker issue a warning of the form "Variable X is unbound in function F."

Decisions (4dec07)

Based on the several entries of today's date that follow, plus other entries further down the list, and in the interest of simplification, here are what I hope are some definitive decisions:

a. *possibly* eliminate *all* abbreviated keywords (except pre/post); specifically: ax, eq, func, is, obj, op, ops, precondition, postcondition, thm, val, var

b. *alternatively*, eliminate *all un*abbreviated keywords (except axiom); specifically: ax, eq, func, is, obj, op, ops, precondition, postcondition, thm,

c. get rid of functions altogether, leaving only `op` as the keyword

d. get rid of the separate executable expressions, retaining assmnt (with new 'set' syntax), foreach, and while, thereby eliminating `if-then` as a statement and `return`. This leaves exactly three non-functional constructs -- set, foreach, while.

e. forget about typename field access, once and for all

f. change the use of "`op`" in a parts spec to "`lambda`" or perhaps "`function`"; it would be nice not to have any keyword at all for a function type expression, but a quick hack in parser.y adds 2 s/r and 4 r/r conflicts; we may be able to get it to work, but I don't know at this point (obviously don't do this if we go with the get-rid-of-funcs idea)

g. replace '<' with '>' as extends abbreviation

h. replace the term "atomic type" with "primitive type"

i. replace the term (and keyword) "attribute" with "property"

j. get rid of sym lits

k. forget about "`Object`" as the top type

l. go for type vars fully, including use in generic obj decls and op signatures

m. the use of '`,`', '`and`', and ' ' should be fully interchangeable in all list contexts, insofar as LALR allows it

A Not-Yet-Fully-Explored Problem with Typename Access to Fields (4dec07)

What do we do about cases where the name of a tuple field or parm is not a plain ident, as in

```
ManagerRecord > PersonRecord = Employee*;
InterspersedStuff = (Tag or Text)*;
```

?

I believe the answer must be that typename field access can only work when the type of a component is a plain ident, which does *not* include a type var. This is a bit of a duh, in that the designation type*name* field access means that we're referring to the field by it's *named* type, not some type expression.

I'm not sure at this point if this realization is a potential nail in the coffin of typename field access. I think it might well be. In reading again the 23apr03 entry, I'm thinking it's not really as compelling an example as it seemed at the time. Further, having a good IDE, with tuple component completion, could go a long way towards lessening the need for typename field access.

So let's do an explicit pro/con analysis for typename field access:

Pros:

a. simple, in particular to the novice, e.g., the intro SE student

b. can be cleaner and shorter

Cons

a. violates simplicity principle
b. notationally violates the use of type names in expressions, even if it's a shorthand that does not violate the underlying semantics
c. does not apply uniformly to all types, specifically, to tuples with non-ident field types
d. gets complicated for tuples with two or more fields of the same type
e. a decent IDE, including specldoc inclusion of show/hide names button, can go quite aways towards lessening the benefit of typename field access

So far, it looks pretty clear that the cons have it.

No Topmost Object Type (4dec07)

If we go for nicely parameterized objects, I don't think `Object` as the "top" type is necessary. Not having it makes us more functional "feeling" than OO feeling, for what that's worth. Further, it avoids the weak typing hole exemplified in `newer-inputs/min-max.fmsl`.

Possibly Eliminate obj and op keywords Altogether (4dec07)

The idea exemplified just below, of no keywords at all, is just fine. I can even live with using angle brackets for parameterized objects. As far as keyword elimination goes, what just might work best of all is to eliminate the abbreviated keywords, e.g., "`obj`", and "`op`". This seems to be a reasonable compromise between retaining keywords for clarity, if the user so chooses, versus simplifying by eliminating excess keywords.

And with a decent IDE, the shorter keywords aren't really all that useful. The keywordless forms are good for on-the-board use and quick typing, while the full-length keyword forms will work fine in a template-driven IDE.

The other nice thing about advertising the keywordless forms is that it makes grammar definitions just a normal thing, without doing something funky to allow grammars as a special case. In the "I can even live with it" file again, I'd put using just plain '=' as the LHS/RHS separator, instead of having "`::=`" as yet another "is" keyword.

Here's an example of full-blown keywordless and variableless defs:

```
PersonRec = Name and ID;
DB = PersonRec*;

Add(PersonRec, DB)->DB
    pre: PersonRec.Name != nil and not (PersonRec.Name in DB);
    post: NoJunkNoConfusion(PersonRec, DB, DB');
end;

NoJunkNoConfusion(item:?T, in_list:?T*, out_list:?T*) =
    item in out_list
        and
    forall (item' in_list) item' in out_list
        and
    #out_list = #in_list + 1;

GenericList<?T> = ?T*;
```

Note here that:

- we can't use regular parens for parameterized types, since they'll be ambiguous with keywordless op defs;
- when field or parm types are not plain idents, names *must* be used; a type var is *not* considered a plain ident.

With this, I still think it's fine to say that 'and', ',', and ' ' are synonous ways delimit list items.

An advantage of having no obj/op keywords at all is that allows them to be used as var names, avoiding the possibly confusing error when using a keyword as a user-defined ident. However, I think this is a small enough advantage that retaining obj/op keywords is highly arguable. An advantage with this is that it can help a definition read more cleary.

Regarding (still) allowing coponent and parm names, it can be argued on the basis of style, particularly if one wants to make expressions less verbose-looking, as in the following equiv to the above:

```
PersonRec = n:Name and id:ID;

Add(db:DB, pr:PersonRec)->db':DB
  pre: pr.name != nil and not (pr.name in db);
  post: pr in db';
end;
```

And more importantly, we *must* have names when field types are not plain idents (see "A Not-Yet-Fully-Explored ... (4dec07)" item above).

In the end, as long as we don't make things absurdly complicated and/or sacrifice understandability severly (if at all), having optional syntaxes will be fine. And we can have a smart(er) syntax checker that warns of inconsistent notations, if the user wants to turn on such warnings. Call in "notational lint" mode.

Possibly Eliminate Non-Abbreviated Keywords, For Simplicity (4dec07)

Candiates to get rid of include "`object`", "`operation`", "`function`", "`operations`", "`precondition`", "`postcondition`".

Problems with this include:

a. text book presentation using full word "`object`" may look better than "`obj`"
b. specs arguably look better with the fully spelled-out entity keywords
c. I think there are more cases where abbreviated keywords are a pain than the non-abbreviated ones
d. while I don't much care for the fully spelled-out keyword "`variable`" compared to "`var`", I can live with it for overall consistency

Op Equivalent of 23may02 Tuple Notation (4dec07) E.g.,

```
op O(A, B) -> C
    pre: A < B;
    post: C  = A + B;
end O;

op P(D, E) -> E
    pre: D < E;
    post: E' = D + E;
end P;
```

Possibly big problem here is that we're using type names directly in exprs. We could do this:

```
op O(A, B) -> C
    pre: in.A < in.B;
    post: out.C  = in.A + in.B;
end O;
```

but I think that's fucked up.

But maybe as long as it's clear what we're doing here, referring to vars by their type names could be OK. We'll just have to do some definitive examples to be completely sure.

Axiom Syntax, Yet Again (1dec07)

Despite the 26nov07 thoughts on the matter, I think having axiom (and theorem) names is a prefectly reasonable thing to do. It'll provide a consistency with other entity defs. At least that's the thought for the moment.

Syntax Rationale (1dec07)

'=' binds a name to an expression. Specifically,

- in an `object` definition, '=' binds an *object name* to a *type expression*
- in an `operation` definition, '=' binds an *operation name* to an *unevaluated computable expression*
- in a `value` definition, '=' binds a *value name* to a *compile-time-evaluated computable expression*
- in an `axiom` or `theorem` definition, '=' binds a *formal declaration name* to an *unevaluated computable boolean expression*
- in an initializing `variable` definition, '=' binds a *variable name* to a *run-time-evaluated computable expression*
- in a `let` expression, '=' binds a *variable name* to a *run-time single-assignment computable expression*
- in a `set` expression, '=' binds a *variable name* to a *run-time-evaluated computable expression*

Yet More to Do to Finish Things Up (29nov07)

Define fully the semantics of relational operators on all types. To whit (for starters, at least):

- for lists, the lenghts of both operands must be the same, and if so, each element is compared
- for tuples, operands must be strongly compat, not just subtype compat; this ensures that both operands have the same number, type, and position of fields; if this is the case, fields are recursively compared
- for unions, the operands must be strongly compat, which means that the fields of the union type must all be strongly compat

If I'm not mistaken, we've used the term "equivalent" to mean strongly compat. Whatever, we need to be precise about all of this, both in the ref man and the SpecL spec.

Think through the Co-existence of Generics and Type Vars (29nov07)

4dec07 Update: Given that we're going with angle brackets for generics, the revised version of the example that follows is this:

```
obj GenericList<?T> = ?T*;
op Op1(gl:GenericList<integer>, i:integer) = gl + i;
op Op2(l:?T*, x:?T) = l + x;
```

and the discussion that follows is now obsolete, insofar as it deals with the use of parens in the syntax of generic type instantiation. It's retained here for historical and rationale purposes.

If it's not been stated explicitly (enough) so far, this needs to be done, to ensure that the co-existence is in fact doable. E.g.,

```
obj GenericList(T?) = T?*;
op Op1(gl:GenericList(integer), i:integer) = gl + i;
op Op2(l:T?*, x:T?) = l + x;
```

The on-the-fly generic instantiation seems not to conflict with a construtor invocation, since the former is used in a declaration context, whereas the latter is used in an expression context.

Adding State Machines to SpecL (29nov07)

Sparked by need for a state machine definition as part of FIPS 140 compliance, I'm thinking it might not be that tough to add state machine syntax and semantics. The graphical forms are a state-transition table and FSM graph.

Some would-be syntax:

```
(* Input symbols: *)
a; b; c;

(* Output symbols: *)
d; e; f;

machine M1
    states: S0, S1;
    transitions:
          --> S0,             -- Initial state
          S0 --a--> S1,       -- Goto S1 from S0, on input 'a',
          S1 --b--> S2,       -- Goto S2 from S1, on input 'b'
          S0 --c/d--> S3;     -- Goto S3 from S1, on input 'c', producing output 'd'
  description: (* ... *);
end M1;

state S0
  inputs: a,b,c;
  outputs: d,e;
  description: (* ... *);
end;
```

The presence or absence of inputs and outputs determines the type of machine it is:

- if all states have inputs only, with no outputs for any ops, then the machine is a recognizer
- if all states have exactly one ouput, it's a Moore transducer
- if one or more states have more than one ouput, it's a Mealy transducer

If we have the gumpshum, the type checker can perform the following (types of) checks:

- determination of NFA versus DFA
- determination of recognizer versus Moore machine versus Mealy machine
- malformedness, including:
  - *o* no start state, if appropriate
  - *o* no end state, if appropriate
  - *o* disallowance of non-opaque objects as symbols
  - *o* disallowance of outputless state of one or other states has outputs (or default to nil ouput?)
  - *o* ?others?

We may want to consider the keyword "`symbol`" as a equivalent way of defining opaque types. I.e.,

```
symbol x;
symbol a,b,c;
symbol d
    description: (* ... *);
end d;
```

is equivalent to

```
object x;
object a;
object b;
object c;
```

and `symbols` are disallowed from having a components attribute.

HOWEVER -- Given that opaque objects can be declared as shown at the top of the preceding example, i.e.,

```
(* Input symbols: *)
a; b; c;

(* Output symbols: *)
d; e; f;
```

I really don't think the "`symbol`" keyword is necessary, at least for the state machine context.

Yet Moron Short-Form Object Defs (26nov07)

Am I smoking something, or is '<' backwards for a UML-like mnemonicette for inheritance. I.e., shouldn't it be '>', so the inherited-from object is in the pointy end of the arrow? E.g.,

```
object EmployeeRecord > PersonRecord
    = WageScale and EmployeeStatus;
```

versus

```
object EmployeeRecord < PersonRecord
    = WageScale and EmployeeStatus;
```

or even

```
object EmployeeRecord > PersonRecord
    <> WageScale and EmployeeStatus;
```

Also, we should consider allowing extends and '=' to be interchanageable in the order of an object definition, as in

```
object EmployeeRecord = WageScale and EmployeeStatus
    extends PersonRecord;
```

versus

```
object EmployeeRecord extends PersonRecord
    = WageScale and EmployeeStatus;
```

Rethinking Axiom Naming (26nov07)

I'm not sure there was ever a big clamour for naming axioms, and so, given the potential unclarity of the name/axiom-body separator syntax (i.e., ':' versus '='), I'm not really sure we need to have axiom names, after all. What I'd like to see is a complelling example were axiom naming is useful.

The arguments against it include:

a. funky syntax
b. no seeming way to "invoke" or use an axiom by name in a spec, meaning the naming is only useful for (humans) talking about a spec
c. as far as browsing goes, we could have the checker automatically enumerate the axioms, giving them module-qualified names of the "*`M`*`.An`", where *`M`* is a module name (including the default "`Main`"), and *`n`* is a unique gen-sym'd integer, starting at 1, for each module; then the browser could take the viewer to the appropriate source code, when the user clicks on the axiom name in the browser
d. using the new javadoc-style comments, the comment immediately preceding the axiom definition would go in the data dictionary for that axiom; an entry in the data dictionary for an axiom is of the form

   ```
   Name (auto-gen'd)    Expression    Description
   ```

   which is nicely analogous to the data dic entry for an object, viz.,

   ```
   Name (declared)    Components    Description
   ```

How Type Safe are Unions?? (21nov07)

Do unions of incompatible types imply runtime type checking? Reread Tennent and look at functional languages about unions, including ML (datatypes), Haskell, and Z.

As we've at least hinted at before, I think I like the idea of having different levels, or strengths, of type checking. At the stongest level, we could say that un-overloaded equality is not defined for unions of incompatible component types. For sure, look again fully at what ML does to require that functions over datatypes account for all alts of an ML `datatype`; see, e.g., 530/ml/lispval*.ml. The warning message from the ML compiler is "match nonexhaustive", the specl analog of which is the preceding idea of "weaker" type checking.

Type Var Naming (21nov07)

Figure out if type vars should have the '?' at the beginning, end, either, or one or more times anywhere in the identifier string.

Also figure out whether type vars should be required in defining parameterized (aka, generic) types. See, e.g., the versions with and without the syntactic use of type vars in newer-inputs/parameterized-types.fmsl.

Also figure out if just plain '?' is OK as a type var, as long as we don't care about neighboring types constraints. However, it may be the case that '?' outside of the context of an ident may cause syntax problems. Need to check it out.

Also figure out if we should limit '?' in an ident to type idents, i.e., have separate lexical/syntactic categories for ident and type_ident.

Yet Moron Posssible Alignment with UML Terminology, and Related BS (15nov07)

Re. the rationale for the term "object" instead of "class", I think we can rationally argue that specl objects are not really classes in what might be called the widely-conceptalized sense, for at least the followning reasons:

a. operations do no belong to objects, in the strongly object-oriented sense of UML
b. specl objects can in fact be considered to have aspects of both UML classes and UML objects, in the sense that specl objects can contain concrete values
c. fundamentatlly, specl objects are most closely related to UML/OCL `types`

It is clear that English word "object" is highly overloaded when used in SE circles. The sense most closely aligned with its use in specl comes from the phrase "objects and operations", used in context of software requirements analysis and modeling. Yadamotherfuckingyada.

Extending Primitive Types (12nov07)

Despite what it says below in the 27oct07 item, I think "primitive" is in fact a better term that "atomic". It's akin to the "attribute" versus "property" discussion. I.e., since I don't see any particularly compelling reason to favor "atomic" type over "primitive" type, we can go with the latter on the grounds of overall clarity.

Now to the question at hand -- can we extend primitive types. I'm not sure we've come to a definitive conclusion on this, but we clearly need to. What needs to be done is a scan of items in this file, and elsewhere, followed by a DECISION. You know what one of those is, right motherfucker?

Moron Posssible Alignment with UML Terminology (10nov07, cf 4nov07)

One possible, seemingly benign change in SpecL terminology (yes, I think I'll start calling it that now) is changing specl "attribute" to "property". This avoids confusion with UML's use of the term "attribute", and there is some historical precedent for the term "property" in spec lang contexts such as this.

More Syntx Fiddling (6nov07)

4dec07 Update: We will in fact use '=', as discussed in the "Syntax Rationale" entry of 1dec07.

With the full advent of '=' instead of 'is', we should consider if it makes better semantic sense to have the tuple component initializer be ':=' instead of '='. What brought this on was the odd-looking one-tuple definition in newer-inputs/const-components.fmsl:

```
obj OneTupleOfIntegerInitializedToTen = integer = 10;
```

with the thinking that

```
obj OneTupleOfIntegerInitializedToTen = integer := 10;
```

looks (somewhat) better. The question to be answered is if component initialization is more like an equality definition or an assignment.

I just had a somewhat sickening thought about possibly (re...re)changing the syntax of object declaration (back to) allowing (having) ':' be the (only) separator in a short-form obj definition. A quick hack to the grammar revealed that it appears to work.

But fuck it -- I think '=' is just fine for binding a type name to a type "value". We can go on about this, but I really don't think we need to.

Generics (4nov07)

4dec07 Update: With the use of '>' instead of '<' as inheritance sugar, there is no extra s/r conflict when using '<' ... '>' to bracket genric object parms.

Consider strongly replacing the current `where` attribute stuff (aka, kludge) with a more conventional syntax, as illustrated, e.g., in newer-inptus/parameterized-types.fmsl. A syntactic issue is the conflict between '<' as a type parm bracket versus the inheritance symbol, which in fact does cause an s/r conflict in the current grammar, when just hacked in. There are a number of syntactic options, including using a different bracketing, or changing or eliminating '<' as inheritance sugar.

Further thought on this syntactic issue is that regular parens should be fine for parameterized types, as shown in the second example in the `parameterized-types.fmsl`, and the commented out RHS in `obj_heading`, dated with today's date (4nov07). We need to make sure it all works out in terms of type refs, but hopefully it's OK. There's someting to be said for regular parens instead of angle brackets, on at least two accounts. First, regular parens are consistent with the terminology "parameterized types", akin to the notation for parameterized functions. Also, using regular parens avoids the introduction of additional bracketing syntax, when it is arguably unecessary.

One potentially very good thing about this more explicit form of type parameterization is that it may well at least partially address the more "featureful" form of composition in UML 2.1, vis a via UML 1.5, as discussed in `../related-work/uml/general-info/UML-2-comp-model.pdf`. This was rather worrisome on first reading, but generalized parameterized typing seems to go in this direction, in that the type parm can be used in relational attributes, in addtion to just the components that (seemed to be) the case for where instantiation. More thought is necessary here, including the *redefines* and *subsets* association constraints that are most likely related to all this.

Re. the semantics of generalized type parameters, my initial thought (hope) is that it can be very close if not identical to the current semantics of where clause instantiation. This obviously needs to be worked out. We of course need to study (and understand) fully Java generics.

Posssible Alignment with UML Terminology (4nov07)

In looking some more at the UML specs, the thought occurs, again, to consider using terminology that's more consistent with UML, e.g., "`class`" instead of (in addition to) "`object`", "association(al)" instead of (in addition to) "relation(al)". What I would very much prefer is to spell out the terminological diffs between FMSL a UML, and rationalize why they exist. And I think this preferred course is in fact quite doable.

Enum Trouble in River City (4nov07)

Whither `next` and `prev` ops on enum values? It's a bit kludgy, but perhaps we can say that if *all* union components are of type "the T x", for some type `T`, then there are `next` and `prev` ops defined. This implies there's an order to union (and tuple) components, which in fact there is, per the ML-like notion of `#n` to access the nth component. What we're saying here is that if a union (and, what the heck, a tuple) has components of the same type, then there are `next` and `prev` functions available.

We of course need to define "same type" formally, and there are some subtype and type compatibility things going on here we need to deal with. E.g., if all of the elements of a union (tuple) are compatible with each other directly or indirectly, there are `next`/`prev` ops. *Indirectly* compatible means something like "compatible with other type", as in forall `i` "the T x_i" are compat with `T`.

What we might be able to do is generalize this to all unions and tuples being fully indexable, in the normal square-bracket way. What we do is make the co-arity of the index op be the union of the component types, modulo union type simplification. This notion of union *simplification* is related to (or may subsume) the notiion *indirect compatibility* mentioned in the preceding paragraph. E.g., "union of T or T or T" simplifies to "T"; "union of the integer 1 or the integer 2 or the integer 3" simplifies to "integer". Sounds promising (but so do a lot of other motherfucking things).

Moron Relational and Valued Attributes (4nov07)

Allow multiplicity in Relational Attributes. See, e.g., newer-inputs/attr-multiplicity.fmsl.

Make the functional basis for relational values be `ref`.

I don't remember right now why we currently allow attributes to have general expressiions as values (see the parser syntax for `obj_attribute`, which includes `expr` as a RHS alternative. Given that we seemed to have gone whole-hog into concrete-valued objects, we don't really need this alt any more, and it causes a reduce/reduce when we added the syntax for relational multiplicity shown in the newer-inputs example cited just above. So, we should get rid of it, if at all possible.

Cleaning Up Syntax and Semantics of Const and Init'd Components (3nov07)

As noted in the LOG entry of this date, the "possibly starred" biznis in parser.y seems rather kludgy. As I recall, it was done incrementally to allow stars in signatures.

With the coming of refs, we need to revisit this area, and see if we should go a route like "possibly starred and/or reffed", or what the fuck. We may just want to rethink the ML-style of auto-tuple args, though as I recall this had some potentially serious problems, like the concept of a truly multi-arg function in a dataflow diagram. Anyway, we need to get closure on this.

Related to this is the idea of the difference between constant-valued tuple components, versus initialized tuple components, as illustrated in the following examples:

I think it's pretty clear at this point, if it has been so (and discussed) till now, that the verbosely mnemonic object names of this exmaple spell things out. Now we must make sure this shit is spelled out in the ref man, and whereever the fuck else it should be.

Regarding the fabled "ref man", I think it's time to have an official two- or maybe even three-pass version, akin to the "Gentle Introduction to Haskell" versus the full Haskell ref man. I think we can readily motivate the gentle intro, including including some phrasiology like "But can I do this ...", a preliminary lists of subheadings for which includes:

- Can I define constant data fields?
- Can I define statically initialized data fields?
- Can I define default values for operation inputs?
- Can I define generic objects, as in Java generics or C++ templates?
- Can I define an object-oriented model?

- Can I define something that looks like a relational data model, as in an ER diagram or the equivalent in UML?
- Can I model something that looks like a grammar, including an attribute grammar?
- Can I define general UML-style associations and multiplicities?
- Can I define an ontology?
- Can I test a model in some way?
- Can I prove things about a model?
- Can I execute a model?
- Can I model sequential program flow in some way, short of executing it?
- Can I model a state machine, petri net, or UML activity diagram?
- Is there an easier way to access tuple compnents, other than having to name every component?
- Can I define global variables, even though I shouldn't?
- Can I specify the behavior of an operation with "pseudo code", instead of preconditions and postconditions?

Summary of Syntactic Sugar, Including Auto Ops (30oct07)

I think we should fill this in here, and include it in some form in the ref man. Here's the list so far:

1. auto-deref-on-field-select-of-ref-to-tuple
2. auto-unbundle-n-tuple-as-arg-to-n-ary-op
3. auto-bundle-n-args-into-n-tuple-for-1-ary-op-of-n-tuple-arg
4. auto-gen-constructor-ops-for-objs
5. (possibly) auto-gen-of-new-heap-object-on-binding-of-constructed-value-to-ref
6. allow type names as tuple field names, with appropriate disambiguation
7. `#n` form of tuple field access
8. operator overloading
9. all of the syntactic alternatives and short cuts

The Point of "obj EnumLit = string" (30oct07)

With all the thinking we've done, the question for a newbie might be ''Why allow both "value X = const-val" and "object X = const-val"''. A decent answer for allowing the latter is to say that it's a degenerate case of allowing a type to have one or more constant values as components, without creating one or more special-case exceptions for what is and is not legal. Moreover, as long as there's no harm in allowing this, things are just fine.

The benefit of allowing values in general to be object components is it provides a natural and straight forward way to define enumerated types, as can be done in most PLs. In addition, one can easily model other forms of enumerations, all stemming from one basic idea.

More on Ref Type (29,30oct07)

Given that both UML and ML have refs, I think we should consider seriously adding them to FMSL (cum SpecL). The ML syntax and semantics looks a bit funky, so we could go with something like this:

a. Examples:

```
obj A = integer;
obj B = boolean;
obj C = string;
obj ARef = ref A;
obj ABC = A and B and C;
obj ABCRefs = ref A, ref B, ref C;
```

```
op Op(a:A, b:B, c:C, aref:ARef, rt:RefTuple) = (
    a = 10;                -- normal int var and val
    aref = new A(10);      -- ref var and heap val
    aref = A(10);          -- equiv OR replacement for the to preceding;
                           --   see discussion below
    aref = new A();        -- ref var and heap val, uninit'd
    @aref = 10;            -- deref to access "ref to int"
    (@aref).A = 20;        -- deref then field select
    afef.A = 20;           -- equiv to prev expr, courtesy of
                           --   auto-deref-on-field-select-of-ref-to-tuple

    aref = new integer()   -- ERROR: new can only be applied to user-defined types

    let a = 100;           -- single assignment
    set a = 200;           -- mutating assignment
    a := 200;              -- syntactically sugared mutating assignment

-- a bit more syntax
obj X = rt:ref T and lt:T*;
)
```

b. `ref` as a keyword can only be applied to a named type, as is `extends`.

c. In the area of shallow versus deep equality, hopefully we can eliminate the need for two operators. Viz., for two ref vars `aref` and `bref`,

```
aref = bref        -- shallow equality
@aref = @bref      -- deep equality
```

d. If we do things right, we may be able to create heap values without an explicit new. However, for clarity, it may be appropriate to retain the `new` operator, just to make it completely clear when and when not refs are created. This said, here are a some (the?) possibilities in this area:

   i. Say that a constructor call always returns a ref, and require '`@`' to stick it in a non-ref var.

   ii. Say that a constructor call always returns a non-ref, and require '`new`' to stick it in a ref var. (NOTE: This is probably the best alternative.)

   iii. Say that a constructor call returns a ref or a non-ref, depending on the context, i.e., what it's bound to. In this case, the only indication that we have a ref value is what it's bound to, and the seemingly only reason we need '`@`' is things like deep equality (but hey, "things like" does not really match up with "only reason", but whatever, you get the fucking drift).

   The good thing about the last approach is that it fits pretty darn well with modeling, as opposed to programming. I.e., what we care most about is allowing refs to be used intuitively in a model, including its pre- and postconds. Given this, and given the idea of trying to define analytic, aka, non-constructive specs as much as possible, the transparency of using on not using `new` is not our focus. Hence, we could legitmately say something along the lines of "Hey, dudes, when you start hacking with constructors, you need to know what the heck you're doing, and be aware of the fact that when you bind a value to a ref var, the value goes on the heap." But having just said that, I still may be inclined to leave in the `new` opeartor, again for clarity's sake, and since its use is not really much if any of an inconvenience.

e. The 17dec04 item below talks about possibly eliminating auto-gen'd constructors, based on a lack of need for them. However, with this new ref stuff, I think they're in fact necessary. The reason is that we need a way to construct built-heap values explicitly, and if we disallow anything but user-defined types on the heap, the (auto-gen'd) type constructor makes sense, if not being required.

f. So, if we've not spelled it out fully by this late date, we'll say that for every user-defined obj type T, there are these overloads of an auto-gen'd constructor op created:

   i. `T()` -- parameterless constructor op, leaving all fields uninitialized

   ii. `T(T)` -- full initializing constructor op (but see just below)

   Re. the full-init version, see the discussion in `new-inputs/fiddling-with-auto-gen-constructors.rsl` and `new-inputs/uto-gen-constructors`, the important gists are auto-unbundling to make constructor invocations slightly more wieldy, and the need for "holes" in tuple constructors for tuples with constant values.

But wait on the following "but wait". What's emerging here are at least a couple ideas:

a. The "pure" subset, or "core" of SpecL, that we can describe and motivate in the ref man. It would be cool if we could define this as simply as "no refs and no sets" and possibly "no constructors". And since loops are useless without sets, it leaves them out too.

b. A potentially clearer picture of why refs are hard to verify. The deal is that when we create a `new` value, we're adding it to a pool, aka heaplet of values of its type. I'm thinking there may be an equivalent "no junk, no confusion" rule for ops that modify heap values that we must state that the modification has no effect on anything in the value's heaplette. Hmm, pretty interesting, this, and we may on to something.

BOGUS: But wait just a farging minute. Java has no explicit refs in it, and we seem to be able to do just file with it. But then, there's the `setf` stuff in Lisp, that we really should consider. Hmm, this needs plenty of thought. :SUGOB

Death (at Last) to Sym Lits? (28oct07)

OK, here's what I think are the reasons we want to keep sym lits, and I'm getting to the point where I don't think the reasons are strong enough to keep them:

- the value of an opaque type -- *but we can argue that opaque types don't deserve values -- that's what opaqueness is all about; e.g., we can define fieldless classes in Java without ever worrying about the fact that they don't have some literal value notation for them.*
- for non-terminals in the RHS of grammar rules -- *but hey, I think double-quated strings work just fine for this, and might even be better in some sense (need to clarify why I think this)*
- as a more "accurate" model of constant enumeration literals -- *but wait, is 'Monday' really more "accurate" in any sense than "Monday"??*
- abstract versus concrete syntax -- *but I think this can get taken care of by opauque types themselves, without the use of opaque values, i.e., sym lits*

Now that we have (or just about have) our good clean semantics for concrete values as object components, I think we can hop in the way-way-back machine, to the era when I seemingly naively thought about how potentially cool it was to define an enumeration with just plain strings, as in

```
object Sex = "Male" or "Female";
```

I mean, it says what we want, we can have a strong-typed semantics for it, and we don't need to mess around with several other ways to define the same thing with opaque types and/or sym lits.

Bottom line -- let's go for some simplicity here.

One More Time with Generics and a Built-In Topmost Object Type

It would appear what we have with generics is a reasonable deal for a modeling language, in that allows some useful things to be defined in terms of overloading and inheritance.

We've thought in the past about somewhat gratuitously throwing in the Java-style *`Object`*. I don't think we can just do this without doing one of two things:

a. Milner-style type inferencing
b. opening the door to dynamic typing, via down casting

At this point, I think I could go either way here.

Re. type inference, my general inclination is that it's likely to be more trouble than it's worth, if what we're after is a clean spec lang. However, I believe that type inference might be a bit easier in FMSL, given that we do not want ML-style fully typeless op sigs. I.e., even for an op with a fully type-variable signature, we still need to declare explicitly the type of each parm. However, type inference is still likely to be an implementation pain, and the question remains if it's really worth it in the spec-based domain we're going for.

Re. dynamic typing, we just might be able to argue for dynamic typing, as a gateway to an underlying procedural program. In fact, we might have sections in the ref man, after all of the functional sections, with a title like "Dynamic Typing and Procedural Computation", wherein we start out like this:

> "The features described thus far in the manual have defined a fully functional, statically-typed language. In this (and perhaps following) sections are defined procedural features (assignment, looping, refs (maybe)) and dynamic typing features (Object and down casting)."

Lastly, for now, it might just be fine to have *both* type inference and dynamic typing, if we can work out the details OK. But you know, motherfucker, life's getting shorter these days.

The Possible Re-Emergence of Operator Overloading, 28oct07

With the overload-based generics stuff we have going on now, operator overloading might just fit it, even somewhat sweetly. What's come to mind just now (if not in the past), is an example like this:

```
obj GenericDB = GenericRecord*;
op CompareRecords(gr1:GenericRecord, gr2:GenericRecord) = gr1 = gr2;
obj NamedRecord < GenericRecord = name:string;
op CompareRecords(nr1:GenericRecord, nr2:GenericRecord) =
    nr1.name = nr2.name;
```

which could be more conveniently defined like this:

```
obj GenericDB = GenericRecord*;
obj NamedRecord < GenericRecord = name:string;
op "<"(nr1:GenericRecord, nr2:GenericRecord) =
    nr1.name < nr2.name;
```

the convenience being not having to define the `CompareRecords` op at all.

This could be nice, but at the price of op overloading details that may take some time. The underlying hope is that the current generic rules and semantics are effectively unchanged by this style of op overloading.

One issue is that comparison is built-in for all types (remembering that it's tautologically false for all function types). Given this, we may want to limit overloading to the comparison ops, though it's probably not necessary to do this. What we can say is that op overloading has the potential to undermine the clarity of a spec, given that the appearance of an op like "`CompareRecords`" is potentially more clear than just "<", even in the context of record comparisons. And of course there's my frequent peeve about the non-transparency of inheritance altogether, in that the op named `Compare-NamedRecords` would be the clearest of all.

Ah, but a quick check of parser.y shows that strings are still valid op names. So all we gotta do now is update the type checker. (That'll be easy -- chuckle motherfucking chuckle.)

Maybe Already Said, but in Case Not, 27oct07

We should probably disallow inheritance from atomic types.

I was thinking that "primitive type" might be a better term than "atomic type", given that strings really aren't atomic. However, we can in fact think of string as atomic as a *type*, in the sense that it is not composed of other types. The fact that string *values* don't feel atomic is not really relevant to the string type being atomic. And, in fact, we can think of integer values as being unatomic, in that we can decompose them into separate digits using division. So, I'd say, the term "atomic type" is fine.

Maybe Opaque Types === Sym Lits is OK, 27oct07

So the following doesn't sound so bad, and aside from potential confusion of auto-decl of sym lits with opaque decls, might just work out fine:

An opaque type has exactly one value, and that's the symbolic literal of the type's name.

Types as Values, 23oct07, Updated 3nov07

To say in ref man -- Careful:

```
obj TenAndTwenty = 10 + 20;
```

does NOT define TenAndTwenty to be the value 30, but rather to be the two-tuple containing the 10 and 20.

OR, we might just screw the way-old data dictionary syntax that allows '+' as an operator, and go back to only ',' and 'and for tuple creation. This is actually probably a pretty decent idea. Think about it.

3nov07 update: I have thought about it, and I think I do want to get rid of '+' as a component expression operator. Among other things, I recall '+' meaning logical *or* in some dialects of boolean logic, with a multiplication-like "cross" operator meaning something like tupleness.

FMSL Meets Haskell, and Some Other Observations, 23oct07

Did a bit of reading in the Haskell manual and gentle introduction. It would be very informative to do some more Haskell reading, and maybe even refer to it in appropriate places in the ref man.

One particular thing that stood out today was the statement that types are not at all first-class objects in Haskell. In FMSL, it would appear (I'm pretty sure, anyway), that types *are* first-class in the case of union (and thereby) class types. (BTW, if we haven't said it explicitly already, the definition of "class type" should simply be a type that is extended from).

The use of union (class) types that makes them first class comes from the '?.' and '?<' operators, where type names are used as expression operands. This needs to be expounded upon in the ref man.

Yet further musings on opaque types and symbolic literals, 23oct07

Given what appears to be the most recent thinking on this subject, it's not clear if we still want the one-to-one correspondence of opaque types and sym lits. Also, we may want to de-emphasize the age-old way of defining enums using opaque types. (We can't entirely get rid of opaque-type-based enums, since if we have opaque types, we can always make unions out of them.)

A way to do this is to introduce a new atomic type called "*symbol*". With this, the issue pretty much boils down to a binary decision. Viz., is the type of x' "symbol" or "the opaque type x"?

There are a number of questions and/or issues that need to be resolved here, and I'm purposely leaving them unresolved at this particular junction (there's grading to do). Said questions/issues include:

1. The issues of specifier confusing that comes from:
   a. auto-declaring an opaque type whenever a symbolic literal *appears* anywhere
   b. the complementary auto-declaring of sym lit values when an opqaue type is defined
   where such confusion arises when there's a re-defined symbol error for an enum lit or opaque. (But this could most likely be pretty much fixed with a decent error message, that explains why the error happened.)
2. The issue of being able to check the current value of an enum var, as in
   ```
   if (color = 'Blue')
   ```
   as opposed to having to do type interrogation, as in
   ```
   if (color.?Blue)
   ```
3. I don't recall if the current definition/implementation allows the following (I think the def allows it but the imple is not there yet), but either way, having sym lits be part of a "symbol" type would allow it:
   ```
   obj Sex = Male or Female;
   obj Male;
   obj Female;
   ```

```
var s:Sex;
... if (s = 'Male')
```

Just looking (yet again) at this in this context, it looks pretty funky to have a say that there's a symbolic value that's created *with the same name* as an (opaque) type.

4. If we go with the auto-create-sym-lit-on-opaque-type-def, then there's a question of what if any kind of values can be bound to a var of an opaque type. We might want to say that no value but nil can be so bound, but that would leave vars of an opaque type unassignable, and comparable to nothing but nil. That might in fact be OK, if we reason that an opaque type has no discernable value, it's, er, *opaque*, meaning we can't know what its value looks like. This could be a problem if we want to know what type something is by looking at its value, but I think that's probably wrong, since types and values don't mix like this. I.e., we as what type something is with `'?.'` and `'?<'`, not by looking a value that's in a var of a union type. I think this is going in the right direction, but FUCK, we need to get it the fuck nailed down.

5. And while we're up (down?) in here, don't forget that we just re-enabled the stuff in parser.y that allows a type comp expr to contain raw constant values, and we need to fully explain and justify this in terms of the formal type values.

6. For simplification, we may want to do away with the last point, i.e., values in type exprs, given that we argue that we don't do things like range constraints on lists within types. Or, we could (re)instate list range constraints. Anyway, we need to think through once and for all if allowing values in types is (a) OK formally, (b) worth anything. For the latter, we need to dig out / come up with real examples where it's useful.

7. What's going in in the last point is the struggle to allow simple enumerated types as something other than a special case, while not raising a shitload of other theoretical or practical issues.

8. To be a bit more concrete, we can characterize the situation by saying that we can put a value of type "union of (the int 1), (the int 2)" in an "int" var, but we can't ever put it back in a "union of (the int 1), (the int 2)" var.

9. Hmm. What we may just be doing is defining a way to have const data fields without an explicit `const` construct. We can confirm this by thinking about the translation into Java (and UML, if possible) of a type like

   ```
   OneAndTwo = 1 and 2;
   ```

Upgrade to Include "error" as a Distinguished Value, 8may07

In working out ideas in calendar/specification/ideas/duration-bounds.fmsl, q.v., it occurs to me that having `nil` do double duty as both the *empty* value and the *error* value is weak. So, we shoud say that vars can assume three kinds of value --

1. `nil`, meaning empty or unassigned, the latter by virtue of the fact that all the initial state of all not-yet-bound vars is `nil`;
2. `error`, meaning a run-time error occured when evaluating the expression that produces the value to be bound to a var;
3. A non-nil value of the variable's declared type.

Another way of saying this is that every type set contains `nil` and `error`.

Examples of expressions that produce `error` as a value are:

a. index-out-of-bounds error

b. return value of an operation when a precond is violated

c. bound value that violates an axiom

We need to complete this list of all cases where `error` is created, as well as the rules for `error` propogation during expression evaluation, notably does `bool bool-op nil` produce `false`, `nil` or `error`? At this minute, I'd say the latter, but we need to work out the semantics of `if error ...` fully, it most likely being simply that `error` is propogated. Also, it seems that `error` propogation will dominate `nil` propogation, with rules like this:

a. if any one or more operands to built-in operator are `error`, the value of the expression is `error`

b. if no operands are `error`, but one or more is `nil`, `nil` *may* propogate, but we may in fact want to consider propogating `error` in some cases, as in "`x + nil`" = `error` instead of `nil`.

Again, details need to be fully worked out.

Another Bottom Motherfucking Line, Reiterated

Based on the 24may02 and related entries, all opaque types are unique, and incompatible with *any other types*, except `Object` (?`nil`?, if in fact there exists a `nil` type??). Anyway, the important thing is that opaque types are effectively unique. And, as I'm now working a specs for the new dftool, it has occured to me that having opaque types be unique may be pretty handy for dfd editing, in that when a new edge is created between two untyped nodes, a new type is created of the form

`obj` *edge-name* **;**

And to keep things really motherfucking simple, I think we're ready to get rid of "`is`" altogether as a keyword. It'll be a pain to change the test suite, but worth it overall.

NIXED: *[BI] think it's best as is, since, among other things, an ax declaration like this "ax A1 = 1 = 1" looks funkier than "ax A1: 1 = 1"; also, an axiom is not really like an entity that's being bound to a value, but rather a formula that's being labeled. And even if it is, the funky look thing wins out here.*
While we're at it, I think we need to change the syntax of named axioms to be

```
axiom [<name> '='] expr
```

instead of the current

```
axiom [<name> ':'] expr
```

:DEXIN

The REAL Bottom Motherfucking Line

We're going back to the 13nov05 bottom motherfucking line below. The problem with the "*is/Chas*" business is that it's just too fucking hard to use these words so they consistently make intuitve sense, and yet result in a formal definition that works nicely. So, the deal is that we'll sacrifice the intuitive use of "is-a" for an otherwise consistent typing framework. The intuitive background we can refer to is that of standard data dictionary defs, that use "=" as the separator. Also, in my real world, inheritance takes a way back seat to composition, and multiple inheritance takes a way way back seat. So providing a more intuitve "is-a" notation for inheritance at the sacrifice of cleanliness elsewhere, is really not worth it.

It Looks Like is/has May Be a Pretty Cool Deal -- NO IT'S NOT see just above)!

Examples:

```
obj DB has GenericRecord*;
obj EmployeeRecord has Name and ID;
obj SupervisorRecord is GenericRecord has Supervisee*;
obj StaffRecord is EmployeeRecord has Supervisor;
obj Name is string;  -- hmm, is this counterintuitive?
obj ID has number;   -- hmm, how bout this? -- sure
obj DayName has Sunday or Monday or Tuesday or Wednesday or Thursday or
        Friday or Saturday;

obj DB = GenericRecord*;
obj EmployeeRecord = Name and ID;
obj SupervisorRecord > GenericRecord = Supervisee*;
obj StaffRecord > EmployeeRecord = Supervisor;
obj Name = string;  -- hmm, is this counterintuitive?
obj ID > number;   -- hmm, how bout this? -- probably not

obj DB = GenericRecord*;
obj EmployeeRecord = Name and ID;
obj SupervisorRecord extends GenericRecord is Supervisee*;
obj StaffRecord extends EmployeeRecord is Supervisor;
obj Name = string;  -- hmm, is this counterintuitive?
```

```
obj ID > number;   -- hmm, how bout this? -- probably not
```

For a bit of detail, try this. We can explain the two flavors of "`is`" as follows:

a. when an object has exactly one component, you can consider that the object *is* its component; *explain how a one-tuple and an is-a object are semantically the same thing, hopefully coming up with a somewhat better term for "is-a object" vis a vis "one-tuple"*
b. when an object inherits from another object, it is that object, plus it has additional components of its own, thereby extending the object from which it inherits

What these things mean technically from a language perspective is that the `is` clause can syntactically be one of the following:

a. an ident list, probably allowing both comma and `and` to be used as the operator
b. a union or function type, meaning the object *is* one of the union elements, a single list, or a single function type.

The *is* clause can*NOT* be a tuple type; for this, *has* must be used.

Given the extant semantics that one-tuples and is-a types are semantically equiv, *is* and *has* as keywords can be used interchangeably to define is-a types.

I'm thinking "*singleton type*" might be a good name for type defined with is that has only a single name or non-tuple composition expression and no extension with *has*.

Important Consequence of Going with "is" and "has"

For non-tuple types, consider allowing is, particularly for simple enums and function types. What we can be saying here is that any one-tuple type can be defined with "is".

OK, So It's Not "Prim" (15dec05)

Sorry, but "Prim" as a name is (a) a bit too cute; (b) used for a bunch of other things, including "Prim's algorithm" and Source Forge's "Permission Record Information Machine".

"fmsl", on the other hand, shows up nowhere at Source Forge, and is even pretty much nowheresville on Google, with `fmsl.net` and `fmsl.com` still available. And `fmsl.org` is the Franklin Men's Softball League -- pretty much off the beaten path.

Making the Decision to Define Primitives Equationally

Anything we want to consider primitive that's above function invocation, typing, arithmetic, and quantifier-free boolean logic, we'll define equationally. So, here's the list of such things:

a. lists
b. tuples
c. quantifiers

I believe that once we've done this, we can define everything else in terms of non-equational Prim (itself). I need to go out and make sure that this is kosher, but I can't really see any reason it's not. Plus who the fuck is ever going to call me on it anyway?? Well, if goes to Source Forge, it'd be kinda fun to have someone at least notice.

Ideas for V5, aka "Prim"

16jan06 one motherfucking more bottom motherfucking line:

Given the poignant confusion about "is-a" and "has-a" vis a vis RSL's use of "is", I really think we ought to see if we can live with "is" and "has" as the keywords, without '=' or '>' at all. One of the main reasons for '=' is data dict compatibility, but I'm pretty darn comfortable just farging this.

So, for good clean simplicity, one more motherfucking time, try this:

KeywordAttr Name	Meaning
`has`, `components:`	Defines the components of an object. In terms of typing, it binds the type defined by the component expression to an `object` ident.
`is`, `parents:`	Defines the parents of an object, thereby defining the components of an object to be thoses of its parents, unbundled and anded if multiple parents, and then anded to the object's own componnts. In terms of typing, it binds the type "parent-type<1> OR ... OR <parent-type<n> OR parent-types-unbundled and component-type" to an `object` ident.

With this style of defs, we can leave the current type checking semantics in tact, except that we will disasslow inheritance from atomic types, perhaps including opaque types. The latter bit would be consistent with the 24may02 conclusion that opaque types are at the "bottom" of the type hierarchy. However, not being able to inherit from an opaque type may be funky in terms of incremental development, because it precludes defining an object as a place holder and then refining it later. Also, Java (and one suspects other OO PLs) allows componentless classes without any problem.

SO, we need to think through whether to allow inheritance from opaque types. I think it has to be OK. Whether it's OK to inherit from a non-opaque atomic type remains to be determined.

In terms of type rules, whether we do it by spelling or otherwise, I think it's pretty darn clear at this point that any two differenct opaque types are incompatible. I'm not sure if we've thought of this in structural terms, but I think it make sense at that level too, in a perhaps odd kind of way. The deal is that two different opaque types each of no structure, and types with no structure will be considered incomparable, and therefore not equiv (i.e., compat).

13nov05 bottom motherfucking line: here are *THE* keywords and symbols:

Keyword/Symbol/Attr Name	Meaning

`is`, =, `components:` (for object), `body:` (for operation) T} T{
Binds a type to an `object` identifier, an expression to an **operation** identifier, or a value to a `value` identifier; in the case of a type binding, it defines the one or more components of the object; in the case of an expression binding, it defines the functional body of the operation.
`extends`, >, `parents:` T{
Defines the one or more objects from which another object inherits, or operations from which an operation inherits; values do not inherit.
T}	

Symbol (concise) exmaples:

```
obj DB = GenericRecord*;
obj EmployeeRecord = Name and ID;
obj SupervisorRecord = Supervisee* > GenericRecord;
obj StaffRecord > EmployeeRecord = Supervisor and Status;
            -- or --
obj StaffRecord = Supervisor and Status > EmployeeRecord
```

```
              -- or --
obj StaffRecord > EmployeeRecord <> Supervisor, Status
obj Name = string;
obj ID = number;
```

Keyword (verbose) examples:

```
obj DB is GenericRecord*;
obj EmployeeRecord is Name and ID;
obj SupervisorRecord extends GenericRecord is Supervisee*;
                  -- or --
obj SupervisorRecord is Supervisee* extends GenericRecord;

obj StaffRecord extends EmployeeRecord is Supervisor and Status;
                  -- or --
obj StaffRecord is Supervisor and Status extends EmployeeRecord;

obj Name is string;
obj ID is number;
```

You know, it may well be time to dispense with the "has" and "is" business, and just use symbols. The good thing about this is that the sematics of one-tuples being equivalent to type equality is just fine. If we go this root, then only the concise examples below work.

Use `has` and `is` for keywords instead of `is` and `inherits from`, resp. Also, make = and > synonyms for `has` and `is`, with the use of > in particular suggesting the directionality of the UML arrow for inheritance.

Concise exmaples:

```
obj DB = GenericRecord*;
obj EmployeeRecord = Name and ID;
obj SupervisorRecord = Supervisee* > GenericRecord;
obj StaffRecord > EmployeeRecord = Supervisor and Status;
              -- or --
obj StaffRecord = Supervisor and Status > EmployeeRecord
              -- or --
obj StaffRecord > EmployeeRecord <> Supervisor, Status
obj Name = string;
obj ID = number;
```

Verbose examples:

```
obj DB has GenericRecord*;
obj EmployeeRecord has Name and ID;
obj SupervisorRecord is GenericRecord has Supervisee*;
obj StaffRecord is EmployeeRecord has Supervisor;
obj Name is string;  -- hmm, is this counterintuitive?
obj ID has number;   -- hmm, how bout this?
```

Consider using <− for assignment, including for parameter initialization. Regarding parm initialization, there's the interesting issue of default inputs of the form "Untitled *n*", for *n* = 1 ... .

Probably, or at least maybe, get with the program of using "/*" and "//" as comments symbols.

Perhaps replace user-defined attributes with just plain relationships.

Build a GUI editor, with this kind of dialog for objects:

```
=== Object Editor ===
Name:
Components:
Extends:
Description:
```

```
In Module:
V More
Operations:
Equations:
+ -
V More
Relations:
  Name:               Entities:
+ -
...
V More:
Properties:
  Name:               Type:
...
+ -
```

and this kind for operations:

```
=== Operation Editor ===
Name:
Inputs:
Outputs:
Description:
Module:
V More
Precondition:
Postcondition:
V More
Components:
Dataflow:
V More:
Properties:
Name:               Type:
...
+ -
```

and this for modules:

```
Name:
Exports:
Imports:
Entity List:
V More
Theorems:
Axioms:
```

and this for values:

```
Name:
Type:
Value:
Attributes:
```

Keywords

Based on consistent 205 misuse, I'd suggest putting back the singular or "input" and "output", and for consistency, "component". Then, with the good Icon-style more accurate syntax error messages, we can be dandy.

19may05 Op Selection and a Bit on Op Validation

For the CJ types, allow '.' to be used as an op selector, for explicitly-delcared ops. E.g.,

```
obj X
  ops: a,b,c;
end;
obj Y;
op a(X)->X;
op b(X,Y)->X;
op c(Y)->X;
op d(Y)->Y;

op main(x:X, y:Y) = (
    x.a();
    x.b(y);
);
```

I inorder for '.' to be usable as an op selector, the obj being '.'ed must have an declared an op of the name on the RHS of the '.', and that op must have exactly one input of the type of the obj being dotted. And then, the form "*val.op*" means treat *obj* as the (sole) input of type *typeof(val)* to *op*, and do not supply that input in the normal way within the parenthesized list of inputs. All the complicatedness of this reveals just how silly dotted selection of ops really is.

By these rules in the examples above, x.a() and x.b(y) work fine. However, x.b cannot work in any form, because it has no input of type `X`.

A new bit of checking to add to op decls is that an op listed in an obj's ops decls must have the obj in at least on place in its signature. By this rule, it's OK to list c in X's ops, since it has X as an output, even though we can't say x.c, because X is not one of c's inputs. However, d cannot be legally listed in X's ops, since it has X nowhere in its signature.

17Dec04 Note on Values versus Objects (updated 13nov05)

The answer to the "I don't know ..." in the next paragraph is "I do know, and it's now decided". Viz., we can have any mixture we like of type versus value components of an object. As the comment in parser.y explains, we're ready to go with this syntactically, the typechecker just needs to be fixed.

I don't know if we've already decided this, but it would seem that there's a juncture between objects and values at the point where an object is declared with 100% concrete values. However, by the "the" prefix spelling rule, an object can never be used as a value, since its type will never be prefixed with "the". Hence, while the following two defs amount to the same entity in some sense, they are still different because the object version cannot be used as a value and the value version cannot be used as a type.

```
object XO is 1 and "xyz" and true;
value XV = {1, "xyz", true};
```

The object version of `XO` is not in fact particuarly useful, because it can hold only one type of value, namely `XV`. Further, ... (dropped off). However, it's fine to allow it, as a general rule.

17dec04 Update to Auto-Gen'd Constructor Ops

For objects with one or more constant value components, the constructor does not have args for those components. E.g., for

```
obj X is "abc" and i:integer and "xyz"
```

the auto-gen'd constructor is

```
op X(i:integer)->X
```

Fuck, the above applies to unbundling, since the previously-defined deal with auto-gen'd constructors is that their signatures are exactly the objects type, not its unbundled components. The unbundling is a courtesy. What the real issue here is is what the fuck happens to a tuple constructor for constant components. E.g., for the type `X` above, is `{1}` a sufficent constructor, or do there need to be some kind of place holders for the constant components, as in `{,1,}`. I think I like the latter, but it's not currently syntactically legal. Figure it all the you-know-what out. There's a LOG entry of this date to this effect.

And here's another potentially major thought about auto-gen constructor ops -- do we fucking need them at all given structural equiv? I.e., if we have to pass a value constructor in as the arg to a constructor op, why don't we just use the value op directly, instead of having to wrap a constructor call around it? The only reason I can think of is to force the type to be a particular name, but I don't think this is necessary given that this happens at binding anyway. As I recall, the only time we need name equiv is related to inheritance, and I don't see clearly how the value stuff is going to be a problem here. So, check out the following example:

```
obj X = integer and string and boolean;
val x1 = X({1, "abc", true});-- the canonical (and stupid) constructor call
val x2 = X(1, "abc", true);  -- slightly better, courtesy of auto-unbundling
val x3 = {1, "abc", true};   -- OK, but doesn't guarantee x3 is type X
val x4:X = {1, "abc", true}; -- does the trick, without the constructor at all
val x5:X = {1, "abc", 1};    -- just-for-giggles test of type checking;
                             -- fuck me, it fails as of 17dec04, and so needs
                         -- to be FIXED; there's a LOG entry for it
```

I think the comments there say it the fuck all. I.e., if we're a smart motherfucker, we should dump motherfucking auto-gen'd constructors altogether, unless there's some lurking reason to have them around that I've forgotten about, in my reinvention-of-the-motherfucking-wheel way. Fuck me, I need to go confirm this, but I sure the fuck hope this is a nice new discovery to simplify things.

9dec04 Thoughts about Extended Quantifier Forms and Pre/Post Logic in 2 General" 2

Conclusion first, followed by blather. So, here's the part 1 conclusion, about extended quantifier forms. MOTHERFUCKER. It would appear that after all of the bullshit and anguish I've gone through over elseless ifs, the normal truth table definition of implication, aka elseless if, is in fact "if x then y else true". This is because when x is false, "if x then whatever" is always true. Looking at the alternative for of "not p or q" makes this quite clear, since "not p" is true whenever p is false. So, the conclusion to the discussion below about needing to be 100% sure on the equivalences for extended quanitifier forms is that we are now in fact 100% sure. MOTHERFUCKER.

And here's part 2 of the conclusion about elseless ifs in postconds. Since elseless ifs amount to "go true for free cards" in the cases where the if is false, using elseless ifs in postconds is just plain too weak. The deal is that it leaves outputs values unspecified for all of the cases where the if expression is false. This should in fact be covered in the ref man, to say that elseless ifs, in general, and in specific cases as examples, are fundamentally too weak for postconds. What we have to say in general is that a postcond must explicitly specify a condition for all possible values in the range of each output var. What exactly this means fully is part what we have to work out, but elseless ifs in postconds are a significant part of it.

The blather related to the preceding two conclusion paragraphs now starts, from here to the next item.

We need to be 100% sure about the current equivalences that are in the ref man when explaining the extendned quantifier forms, e.g., "forall (x:t | y) p" <==> "forall (x:t) if y then p". The problem I'm worried about right at the moment is the what happens with y is false. I.e., should in fact the equivalence not be "forall (x:t) if y then p else true". I think the fuck so, and I need to figure it totally the fuck out, pretty much right the fuck now.

In general, we may need to come to grips with the one-time-conidered-to-be-an-oddity position of Lois Brady that elseless if don't make sense (in pre/post) logic. This may well be true, particularly for postconds, since the idea there is that we need to cover all possible values for outputs.

Let's push on this a bit. Consider

```
op Op(i:integer)->j:integer
  post: if i >= 0 then j = 20;
```

Given the current rules we've (sort of) got, what about when i < 0? Well, according to the standard truth table for implication, we have this

```
      i      j      i >= 0 j = 20 (i >= 0) => (j = 20)
```

```
-1     19        0     0        0     =>    1 = TRUE
-1     20        0     1        0     =>    1 = TRUE
0      19        1     0        1     =>    0 = FALSE
0      20        0     1        1     =>    1 = TRUE
```

So OK, this really is not bogus per se. What it says is that the only time the postcond fails is when i >= 0 but j != 20. It succeeds whenever i < 0 or when both i >= 0 and j = 20. What this means is that i < 0 is a "don’t care" case as far as the value of j is concerned.

Now, to push on this further, are such "don’t care" cases really sensible in the case of postconds, or do we really have to cover all possible output values in a postcond?

Well, we might say that if a postcond rules out a particular case, then an elseless if might be OK, as in

```
op Op(i:integer)->j:integer
  pre: i >= 0;
  post: if i >= 0 then j = 20;
```

But this is in fact bogus, or at least redundant, since the postcond can never happen if i < 0, given the precond. Hence in this case, the if clause in the postcond is unnecessary.

What I think we’ve arrived at is that "don’t care" cases are in fact bogus in postconds. I.e., we need to have the form be like this:

```
op Op(i:integer)->j:integer
  post: if i >= 0 then j = 20 else j = ... ;
```

since otherwise the value of j is *fully unspecified* when i < 0. But

9dec04 Update to 24may02 -- One More Time with “Does nil = false?”

I think the conclusion there about "x <bool op> nil = nil" may well be fucked up, based on much of what we’ve said in 205 notes, if not elsewhere. Specifically, we need to look at the way we describe what goes wrong with unboudned quantification in sorting to make things completely right. As I recall, the explanation does in fact say that "x <bool op> nil = false".

Constructive Normal Form

A postcond of the form "output-parm = expr".

The general way to get a constructive formulation is if you can solve (easily) for the output var. Presumably this leaves out any quanified exprs as constructive, since I don’t think solving for var inside a quantifier body is possible. Let’s try. I.e., what would it mean to solve for l in the following expr?

```
forall (i in [1..#l-1]) l[i] < l[i+1]
```

Farg, I think it’s pretty clear that it doesn’t mean anything. What I need to do is come up with a bit more precise explanation of the imposibility of solving for l in such a case, based on some reasonable definition of "solve for". Nevertheless, I think the answer is clear that unless we have something like constructive normal form, solving is going to be difficult to impossible.

“Object” as Top

OK, in a lattice-theoretic sense, `Object` is `Top`. *So, how about this as the lattice of Prim types:*

```
                 Object

          tuple union list op

   integer real string boolean opaque

                  nil
```

Exists as Search (13nov04)

See new-inputs/exists-as-search.rsl for the motivation for a new `choose` operator, with the following semantics:

```
(choose (x in l) p)

(lambda (Object, Object*,    ... farg, this needs more thot; and I did not just
                          ... fall off here; we'll hopefully deal with this
                          ... eventually; I added a LOG ref as a nudge

not (forall
```

exists (x:t) p <=> not forall (x:t) not p

OK, I'm not sure on what kind of shakey ground it may put me, but I think I'd like to go with using algebraic specs to define anything we've got that's primitive, but isn't part of pure boolean logic. See the 10nov05 entry in implementation notes for details. The point of this here is that we'll define exists and choose equationally, which should help us deal with the formal "bootstrapping".

Value Spaces (13nov04)

How about this. We can allow the definition of "value spaces" explicitly to bound otherwise unbounded quantification. I.e., we provide some built-in syntax and/or operators to do this.

This can then be combined with the cute trick for testing-based value spaces in the following way. When during testing values are sent to ops, they become part of the value spaces for whatever types they are.

And we might even be able to introduce this cute trick -- everytime a constant of a particular type appears in a spec, it automatically becomes part of that type's value space. In this way, simply *mentioning* a constant value anywhere in a spec adds it to a value space. And this may in fact be a very cute way of avoiding some extra syntax/ops for defining value spaces. We could just say "define some constants" to do it.

Now the just "define some constants" could be tedious for things like integer, but hey, I think we have the solution to this already in the '..' list constructor form. E.g.,

```
value IntegerValueSpace = [1 .. 65536];
value CardValueSpace = [
    { ... };
```

The deal is that the specific value names don't (typically) get used anywhere, but they're appearance forces the internal value spaces to be expanded.

Well, two considerations now come to mind. First, it would probably be nice to do some form of lazy eval of forms like [1 .. 65536], or at least say we're thinking about it when we warn folks that we don't (at present) do it.

A second consideration is one of convenience in defining something like CardValueSpace. It might be nice of we could somehow embed '..' in component values, so as to say that we want multiple values of, say, an integer component without having to define each one explicitly as a constant value.

What this may lead us to is the following idea -- whenever a value is created in any context during spec evaluation, it too becomes a (permanent) part of it's type's value space. This has (potential) ramifications for garbage collecting, in that it seems to imply that we can never garbage collect any values, or at least only garbage collect duplicate copies of values. (And it occurs to me that I'm reminded about why doing the interpreter in Java is a good idea, so we can in fact have garbage collection.)

Anyway, at this point, I think this idea is potentially quite promising, we just need to work out some more details.

The Bloat Goes On

Allow ops to be invoked via "." op, under the following circumstances:

a. the op is listed in the obj's ops list

b. the obj is the first input

Concepts Needed for Process Modeling (15aug04)

Question -- does it make sense to use the '.' op in an output list? It can if we say that it's a short cut for change only the right operand of the '.'. Whether we want to make this happen syntactically is a currently open question.

A pretty serious issue with formal process modeling, even just at the process-step signature level, is that we'll need the highly generic "user input" as a formal input in the signature in order for things to work. I'm not altogether sure we want to open the fully formal can-o-worms for process modeling, as was noted in the Fall 04 version of the process chapter. Anyway, I'll do some more work on chapter 3 and see where things go.

Jass-style Change-Only Semantics

We need some kind of equality operator that says tuple `X` = tuple `Y`, *except* for one or more specific components. The idea of "`let ... in ...`" comes to mind, but seems a bit cumbersome. Also comes to mind is some Icon-like multi-char equality op, but that may be funky, as always. What would be nice is some intuitive-looking operator symbol, with some way to have a list following.

OK, we've come to the syntax that's exemplified in newinputs/except-tst.rsl. One thing worth noting is that we decided to require that the left operand of "~=" be a name, which disallows cases such as the following:

```
forall (x in l) x[i] = whatever except x[i].j = 10;
```

In such cases, a `let` must be used to get the desired effect, i.e.,

```
forall (x in l) (let xi = x[i]; xi = whatever except xi.j = 10);
```

Semantics of List Addition

If it's not already clearly stated, it would appear that the semantics of list addtion are such that adding an element or a listified element have exactly the same effect. Think this through and documentt it fully if not already.

The Semantics of Equality in the Face of the Top Object Type

It seems to me that we most likely need to define the operational semantics in terms of tagged values, since this the only type-safe/meaningful way to compare (with '=' and 'in') two objects that inherit from some other common parent object. The top Object type is the degenerate case, in that equality always type checks correctly statically for two values of type `Object`, but dynamically, equality, specifically v1 = v2, must be defined as follows:

a. if not (typeof(v1) = typeof(v2) or typeof(v1) < typeof(v2) or typeof(v2) < typeof(v1)) then false
b. if (typeof(v1) = typeof(v2)) then *componentwise equality*
c. if (typeof(v1) < typeof(v2)) then *componentwise equality for all common comonents from v2, not including specialized components of v1*
d. if (typeof(v2) < typeof(v1)) then componentwise equality for all common comonents from v1, not including specialized components of v2

The bottom line, as I think has always been the case, is that using inheritance weakens static type checking, and using `Object` weakens the heck out of it. The same goes for union types, and in fact, it's the other way round, since inheritance is defined in terms of unions. And BTW, the definition of inheritance in terms of unions, if it will/can stand in the face of the existence `Object`, will have to go something like this: everytime a new type is defined, the definition of `Object` is dynamically extended with the newly-defined type, in the context of the semantics of the program being type checked/executed.

Reinstating isa as an Operator (14jun03)

Screw the ungrammaticalness of exprs like "`item isa Appointment`". The `isa` operator makes better sense as an expression operator, so as to avoid overloading confusion with `is` as a type definition non-terminal. So, we will in fact get rid of '`?`' as an infix operator, but we'll leave `isa`.

One thing I don't think we got fully explained below (we may have, but whatever) is the difference between `isa` when used as the former '`?`' versus '`?<`' operator. Here's the deal, I'm pretty sure: When used on a top-level union, it's the former '`?`'; when used on an inheriting obj, it's the former '`?<`'; when used on anything else, including a non-inheriting tuple, it's an error. One way or another, we'll always be able to disinguish between an top-level union versus an inherting obj, since the later must always be a tuple. Even if it's a one-tuple, by virutue of inheriting from an opaque parent, it still won't be considered a top-level union for the purposes of '`isa`'. We need to work out the complete details of this, but one way or another we'll make it work, even if it's a bit funky for a seeming top-level union that isn't because it also inherits. The tricky bit has to do with the fact that we can now use a component type name as the right operand of '`isa`', instead of a component name, which means that there's probably no way to treat an inheriting-from-opaque top-level union as a union for the purposes of '`isa` except by using an explicit name or position access operator. This is fine, given that an inheriting-from-opaque top-level union is a rare and funky thing.

More Circles (27may03)

See updates to jan03 item below.

One More Time with Accessing Fields by Typename (23apr03)

This item regards the jan03 and 23may02 items on accessing tuple fields.

The jan03 item concludes that we should nix this type of reference, in part because it's only "a tad" more convenient. Unfortunately, I think the "tad" part is incorrect, based on the experience of adding the weekly-recurring clause to the precondition of the ScheduleAppointment operation, q.v. in schedule.rsl.

The more serious reason to nix typename reference to fields, as stated in the jan03 item, is the inconsistency of this type of ref when it comes to op args. However, at this point, I'm strongly inclined to say that the convenience issue outweighs the inconsistency issue. Given below is a piece of evidence for this inclination, in the form of excerpts of from object defs necessary to support the aforementioned precondtion clause in ScheduleAppointment. The first excerpt uses explicity-named field reference whereas the second excerpt uses typename reference. As can be seen, there's a bunch of clerical, after-the-fact component-name caca required in the first excerpt compared to the second. The actual time spent on the caca is not trivial, since it took me a few passes to get things to compile. The time-to-compile was to some extent based on just hunting down the full ref chain, with or without type names. However, the mere typing of the extra name syntax consumed some valuable time. So, today's conclusion is to bring back typename access to fields, the rules for which are detailed in the 23may02 item.

Here are the excerpts:

```
object Appointment inherits from ScheduledItem
  components: start_time:StartTime and duration:Duration and
      recurring:RecurringInfo and location:Location and security:Security and
      priority:Priority and remind_info:RemindInfo and details:Details;
  ...

object RecurringInfo is
  components: is_recurring:IsRecurring and interval:Interval and
      details:IntervalDetails;
  ...

obj Interval is
  components: weekly:Weekly or biweekly:Biweekly or monthly:Monthly or
      yearly:Yearly;
  ...

object IntervalDetails is
  components: weekly:WeeklyDetails or monthly:MonthlyDetails;
  ...

object WeeklyDetails is
```

```
  components: sun:OnSun and mon:OnMon and tue:OnTue and wed:OnWed and
      thu:OnThu and fri:OnFri and sat:OnSat;
  ...

operation ScheduleAppointment is
  inputs: cdb:CalendarDB, appt:Appointment;
  outputs: cdb':CalendarDB;
      ...
  precondition:
          ...
      if appt.recurring.is_recurring and appt.recurring.interval?weekly
      then appt.recurring.details.weekly.sun or
           appt.recurring.details.weekly.mon or
           appt.recurring.details.weekly.tue or
           appt.recurring.details.weekly.wed or
           appt.recurring.details.weekly.thu or
           appt.recurring.details.weekly.fri or
           appt.recurring.details.weekly.sat
```

versus

```
object Appointment inherits from ScheduledItem
  components: StartTime and Duration and RecurringInfo and Location and
      Security and Priority and RemindInfo and Details;
  ...

object RecurringInfo is
  components: IsRecurring and Interval and IntervalDetails;
  ...

obj Interval is
  components: Weekly or Biweekly or Monthly or Yearly;
  ...

object IntervalDetails is
  components: WeeklyDetails or MonthlyDetails;
  ...

object WeeklyDetails is
  components: OnSun and OnMon and OnTue and OnWed and OnThu and OnFri and
      OnSat;
  ...

operation ScheduleAppointment is
  inputs: cdb:CalendarDB, appt:Appointment;
  outputs: cdb':CalendarDB;
      ...
  precondition:
          ...
      if appt.Recurring.IsRecurring and appt.Recurring.Interval is Weekly
      then appt.RecurringInfo.Details.Weekly.OnSun or
           appt.RecurringInfo.Details.Weekly.OnMon or
           appt.RecurringInfo.Details.Weekly.OnTue or
           appt.RecurringInfo.Details.Weekly.OnWed or
           appt.RecurringInfo.Details.Weekly.OnThu or
           appt.RecurringInfo.details.Weekly.OnFri or
           appt.RecurringInfo.Details.Weekly.OnSat
```

Mother Fucker -- 3mar03

Shit, the processing of imports rears its ugly head again. The LOG file has notes about how we'd like to do them in V4, including some discussion of making them like Java, which concludes with saying we shouldn't try to be like Java, given the difference between what's importable to a module versus package..

So anyway, I'm inclined to go back to a way-old idea that can be characterized as "on-demand" import checking, in order to avoid two possible problems with the "pass-3" style of import checking we're now doing. The benefit of on-demand checking is that it can avoid the following problems:

a. Circularity of imports

b. Importing *all* of a module's exports into a local symtab, when the '.*' for is used in the imports and/or exports.

To clarify and understand exactly how the new style of importing will be done, here's a description of what happens in each pass:

a. In pass 1, done by the parser and sym-aux, all defined symbols (modules, objs, ops, and other named defs) are entered into the symtab, without checking any of their compoents or other attributes. Imports are left unentered, just sitting in their declaration lists waiting for pass 2. What we have at the end of pass 1 are all of the module symtabs allocated, all named entities within the modules entered in the symtabs, and import lists hanging off the modules in purely symbolic form.

b. In pass 2 we can do import processing, ... fell off.

So, superceding all current design and implementation of imports, including what's in LOG, here's a clear and simple import scheme:

a. First, nuke exports entirely.

b. Second, embrace the fact that all top-level modules are part of the top-level global name space, which makes is all visible to each other. Therefore, one automatically has qualified access to *any* module's symbol. I guess this is just what Java does. What's more, Java seems to have solved the transitive type ref problem that Mod2 and FMSL still have, namely that if we import a symbol we don't automatically import the transitive closure of all its component types. What Java seems to be doing to get around this problem is fully qualifying types that come from imported packages. See, e.g., ~/code/java/{p1,p2}/*.bjava. So what we have to figure out is how Java is doing this. Hmm, this may be it:

   i. In a qual'd ref of the following form:

   ```
   M.t.f
   ```

   the type returned is M.F, where F is the type of field f of t, defined in module M

   ii. What appears to be going on is that whenever a module qulifier appears on a type that's the left of of a '.' operator, that module qualification is

   iii. Another thing that we probably have to deal with is the entry of a qualified type directly in the symtab, i.e., with the "." directly in the entered name. If we do this, or even if we don't, it seems that we're going to have to enhance the current resolveIdentType function to include resolution of module qualification in type names.

c. Given the preceding rule, the only reason we need import decls at all is for *unqualified* access to a symbol defined in another module, which we can do a la Java with the "M.*" notation.

d. So what it looks like is that we can in fact do pretty much what Java does now, if not precisely what Java does, to get a much simpler import system.

So, here's the import-checking algorithm we have in mind:

a. In pass 1: enter each imported symbol into the current symtab, flagging it as an import. No definition checking happens at this point, since we may not have yet seen the module we're importing from.

b. In pass 2, we might consider entering all non-conflicing M.x symbols directly in the symtab. Alternatively, we could simple enhance lookup as follows:

   i. Lookup an unqualified type or op symbol in the current symtab.

   ii. If found there, make sure it's not also found in an imported module, and if it is signal a mult-defined error.

   iii. If not found locally, check if it's imported in the form "M.X" for exactly one module, and if so use "M.X". If it's imported in ".X" form from more than one module, complain. (Actually, this is probably the case where we can complain at import decl processing time, since it seems clear that importing the same single symbol from two modules is pointless because it cannot be ref'd unqualified in this case. Need to think this case through, but I'm pretty sure we know what we're talking about.)

iv. If not found locally or as an explicit single import, lookup in all of the module symtabs that have import delcs of the '.*' form.
   • If not found in any of those, signal an undefined error.
   • If found in more than one of thoses, signal a mult-defined error.

From Reading Beckman

Evidently, there is a problem with reference-counting versus mark-and-sweep garbage collectors vis a vis closures. This may well mean that we want to implement the interpreter in Java, using JNI access to the C-built parse tree and symbol table.

NOT -- Let's Just Nix the Component-Ref-by-Typename Business (jan03)

This item regards the 23may02 item on accessing tuple fields. The original title of this item started with "OK -- Let's ...", but now it's "NOT -- Let's ...", which means we've decided to press on with the 23may02 ideas. The reason is that it's significantly more convenient for 205 students, as a visit with one (a 205 student) again made clear today (27may03).

In answer to the "major problem arises ..." comment in the next paragraph, an answer is to do the ML thing, where op args can be treated as a single tuple-valued arg. If we name the args-as-a-tuple value some keyword, like say "`args`", then the problem described in the next paragraph is eliminated. Let's do.

While it might be cute and a tad more convenient, a major problem arises when we want to have the equivalent notation for op args, but we cannot because in the case of op args, there's no qualifying obj name, which means we'd have type names within the exprs of conditions and function values. Given this, I think I'm fine with the following strategy in the reference manual, which is pretty much there already. Viz., have a section after the basic intro to obj and op structure called "Referring to Obj Components" (which is a bit stronger than the current "Names and Types" section). We don't really need this extra strengthening, but whatever. At this point, the important thing is to make peace with the idea once and for all that we need explicitly-declared component names to refer to tuple fields.

In all likelihood, this decision does not affect the recent idea of making the right operand of "is" (formerly "isa") a type name rather than a field name. The deal is that for unions, the type name is used for tag query (as it should be it seems pretty clearly now), and the field name is used to access the value as (of) a particular type.

On "isa" versus "?" as the Type Query Operator

OK, I may just have solved the problem with "`isa`" as a funky operator name, particularly when its right operand starts with a vowel (as in "`item isa Appointment`"). How 'bout we use just plain "`is`" as an operator that applies uniformly to types, never to values. In this way, its use in object defs and also in type query expressions is consistent. Check it out:

```
object X is A and B and C; -- In an object declaration, "is" equates
object A is string;        --  a name with a type expression
object B is integer;
object C is boolean;
value x = {"xyz", 123, true};     -- In a value declaration, "=" equates a name
                           --  with an expression value
function Op(X)->X = ... ;  -- The reason "=" makes sense for function
                           --  values is the construct following the "="
                           --  is in fact a value
object Y is a:A or b:B or c:C;
var y:Y;
... if (y is A) then ...   -- Here, "is" appears in the context of a
                           --  value expression, however it is being used
```

```
                                --  here as a runtime type query, hence the
                                --  use of "is" makes good sense.
... if (y is a) then ...       -- This form is now ILLEGAL, since is should
                                --  apply uniformly and consistently to types,
                                --  not to values.
```

So, with our new-found consistent use of "`is`", can we make the following happen?

```
obj Days is Sunday or Monday or ... or Saturday;
var day:Days;
... if day = 'Sunday' then ...
```

Well, this can work if we make the equality type checking rule go like this:

> one operand is a union type and the other is compat (but probably not subtype compat) with one of the elements of the union. At runtime, the equality evaluation entails the two steps of checking the type, then the value.

What's potentially misleading here is the in the case of enum literals as in the `Days` example, the runtime value check is unnecessary, since there's exactly one value for each opaque type. In an example like this,

```
obj IntOrString is integer or string;
var ios:IntOrString;
... if ios = 1 then ...
```

the need for both the runtime type check and value check is evident. At compile time, the expression `ios = 1` is legal because the type `1` is one of the `IntOrString` union members. At runtime, the evaluation of `ios = 1` involves first checking the current tag of `ios` to ensure that it is `integer`, then performing the normal numeric equalitiy evaluation.

Now, all of this made me think for a sec that we might not need the '.' operator at all for unions, since we can do the injection (or is it projection) now so neatly. However, we do in fact still the '.' on unions in order to bind a union value to an explicitly-declared var of one of the union members, e.g.,

```
op X(i:integer) ...
... X(ios.integer)          -- X(ios) won't work, by the (now) normal rules
                            -- of injection/projection
```

One last bit of syntax -- the use of "is" in an axiom declaration is not really consistent with the above observations. Using "=" as the ax declaration separator would probably be most consistent, but it causes some syntax problems since the right operand of the ax declaration is an expression that can, of course, contain '=' as an operator. Therefore, I think ':' is a reasonable ax declaration separator, while not perfectly consistent with the other uses of "is" and ":", it's fine given the infrequency in of ax decls in the kind of specs I'm likely ever to be doing. Also, upon a bit further reflection, using "=" in an ax declaration wouldn't be all that consistent with the notion of "=" for value binding, since the axiom name isn't usable as a runtime value. Hence, the name binding in an ax declaration is sort of an odd-ball case anyway, so ":" as the separator is probably as good a choice as any, if not in fact the best choice.

Yet, Yet, Yet, ... Again on Unions

I'm not sure it's ever been this clear, so if not, for the zillionth time, here is an explanation of what's legal projectionwise and injectionwise.

```
obj X is A or B or C;
op Main(x:X, a:A, b:B) =
    let x = a;     -- legal because x is already vague, and may hold values
                      of three alternative types
    let a = x;     -- illegal because a can only hold values of type A

obj A;
obj B;
obj C;
```

2jul02, Hmm About Union Types

How about if we dont' *require* that the components of a union be distinct, but just make them that way, in the same way that a set-union operator does. E.g., the type "`string or integer or string`" is equivalent to (and reduces to) "`string or integer`", simply because we automatically throw out the second `string` component. What I'm concerned with here is a potential problem that stems from the combination of structural equiv and inheritance as unions. Viz., in a definition of the form

```
obj Parent is integer and string;
obj Kid1 < Parent is integer;
obj Kid2 < Parent is integer;
```

In this case, the union that defines the inheritance contains two instances of the same type, which by the up-to-now thinking was an illegal union. Something has to happen to make this OK, which could be one (or more) of the following:

a. Say that unions defined via inheritance are special cases that may contain dups (sort of like system-defined idents that can violate normal user-level ident rules).

b. Use name type equiv in some way to mitigate the problem.

What the second of these alternatives suggests is some name-based definition of type *distinctness* or *distinguishableness* that is separatable from equivalence. E.g., we can say that all the components of a union must be *distinct* (or *distinguishable*), and then have a rule that says that any two ident types are *distinct* (*distinguishable*), even if they're (structurally) equivalent. Hmm, this doesn't sound too bad at this point.

1jul02, Possible Reprieve of Sym Lits

There is however the fundamental conceptual appeal of sym lits as distinct from string data in pretty much exactly the same sense of its appeal in Lisp.

Bottom line -- I think it would be stupid to get rid of symbolic literals entirely. What we can do is describe them clearly and rationally in the ref man and say something along the following lines: "The use of sym lits as enum idents is in some sense conceptually more "accurate" than the use of strings for the same purpose, but if the specifier does not care about this, then using strings as enum idents is just fine." The accuracy is concretely evident in the preferred GUI for a particular type of data. Viz., if the preferred GUI is a free-form text-entry area, with a limited set of possible values that are parsed but not enumerated in a combo-box-style menu, then the more accurate representation is a tuple of string values. On the other hand, if the preferred GUI is a completely fixed list of identifiers, then the more accurate representation is a tuple of sym lits. *[And see below about a mixed-mode free-form entry plus combo-box style GUI.]*"

Given this, all of the following defs are legal, and have essentially the same semantics:

definition using sym lits as enum idents:

```
obj Sex1 is 'Male' or 'Female';
```

definition using strings as enum idents:

```
obj Sex2 is "Male" or "Female";
```

definition using opaque types as enum idents:

```
obj Sex is Male or Female;
obj Male;
obj Female;
```

definition using defined string values as enum idents:

```
obj Sex is Male or Female;
val Male = "Male";
val Female = "Female";
```

Today's conclusion: I think I really like the idea of the concrete GUI hinting at the preferred specification representation. But fuck, I was just about to write the following as an example of the specification representation for a string-value combo box:

```
obj Selections is "Alt 1" or "Alt 2" or ... "Alt n" or string;
```

meaning that the "Alt X" strings are the initial given values and the "or string" part deals with a user-entered value. But this is not a legal union type in the latest FMSL semantics, since string subsumes all of the other values. But wait, if we mix and match sym lits with strings, we could have the following, which looks kind of sweet and may (help to) solve the age-old problem of the precise FMSL representation for such things:

```
obj Selections is 'Alt 1' or 'Alt 2' or ... 'Alt n' or string;
```

The reason that this may be a very accurate representation is that it can be paraphrased as follows:

entered by the user.

The potential downside representationally here is that we don't consider `Selections` to be just a plain string with a set of system-suppled default possibilities, but rather a mixed-type union. But hey, this may be precisely the kind of semantics we've been looking for all along.

The I-don't-think-ever-explicitly-defined alternative to the mixed sym-lit string definition is something like the following:

```
obj Selections is string;
obj SelectionsBuiltInDefaults is "Alt 1" or "Alt 2" or ... or "Alt n";
```

The problem we've perennially had with such (assumed) defs is that an abstract op that that takes an input of type `Selections`, with the input tracing to a combo box, needs (seemingly) to take `SelectionsBuiltInDefaults` as well. Either that, or we need to get into the definition of "abstract" view specs, as we did e.g. with Rick Myers, and postulate some op that outputs `SelectionsBuiltInDefaults` to the display, and then some select-from-defaults-list that produces an integer (say) selector that is used by the actual abstract op as an input.

The mixed sym-lit/string type seemingly solves this perennial problem, but perhaps at the price of notational obscurity and complexity. Again, we can say in the ref man that the specifier may chose the representational form that suits her. And as always, we suggest that the same form is used consistently throughout a given specification.

I'll conclude with this thought. It seems potentially a bit hokey, or at best notationally strained, that

```
obj Selections is 'Alt 1' or 'Alt 2' or ... 'Alt n' or string;
```

is illegal but

```
obj Selections is "Alt 1" or "Alt 2" or ... "Alt n" or string;
```

is not. I.e., it's a potentially hokey "convenience" that sym lits can be used to represent combo boxes so "elegantly". We need to think about it (but not too fucking much).

Oh, and one final reason to keep (and use) sym lits -- as we've observed before, they're a good way to represent graphical icons symbolically, where a string overrepresents to some extent.

7jun02, Part 2

We may also want to fuck where clauses hard too. Check inputs/stack.rsl for a decent-looking parameterized type style of generics, that can easily (and clealy) replace the where clause crap. Then again, the idea that we can use type vars directly in a comp expression, without declaring them, may yet have its appeal. Definintely need to think about it.

7jun02

Fuck symbolic literals -- hard. The point is that we'll allow the way-old definition of the form

```
obj Sex is "Male" or "Female"
```

given the fact that we can have non-boolean concrete values in a parts spec. The fundamentally important part about this is that a definition of such a form is *NOT* of type "`string or string`", but rather of type "`(the string the following does not type check, for the indicated reasons:`

```
var s1:string := "Male";
value s2 = "Male";
```

```
op IsMale(s:Sex) = s = "Male";
op main() = (
    IsMale(s1);  -- ERROR s is not compat with s1
    IsMale(s2);  -- ERROR s is not compat with s2
)
```

The point is that the following definition

```
obj S is "xyz";
```

defines S to be of type "`the string string`.

And OK, this has all been made abundantly clear already in the 2jun00 entry, and others below it.

24may02 -- One More Time with "Does nil = false?"

Based on what's said in `exists-as-search.rsl` it seems pretty darn clear at this point that `nil` *does NOT* = `false`. Based on all the formal-semantic error propagation work we've done, plus the fact that a pre-cond-failing op is supposed to return `nil` (*NOT* `false`), we need to have `nil` be a distinct value for all types, including `boolean`. On this basis, making an unconditional equivalence between `nil` and `false` is too much. Now, we may want to make some kind of C-ish equivalence in the context of some boolean ops, but I don't think we even want to do this. I.e., the expressions "`nil` <*bool op*> *x*" and "*x* <*bool op*> `nil` " should both produce `nil`, not a truth value based on some "clever" coercion of `nil` in the context of boolean operators.

I suppose we need to do one last search through all that's been said on this subject, but right how I'm pretty-to-quite happy with the decision.

24may02 -- OK, Here's the Brand New Thing

OK, I just reread the 22mar02 injection and became thoroughly depressed. I think the notion that we can use inheritance to achieve what we're after is dead, when what we're after is genuine statically-typed polymorphism and generics. What needs to happen is the introduction of type variables, of the form we've been musing about for some time.

When this happens, I think we pretty much turn the opaque type business on its head, in terms of how we now see it. Viz., we currently see an opaque type as a topish kind of type from which all other types inherit, with the hope that this scheme makes opaque types act like type vars. As the 22mar02 injection makes clear, this appears to be a forlorn hope. So, what we'll do now with opaque types is put them (back) on the bottom where they belong. They're already there to some extent, in that we have the notion (though I'm not sure it's yet fully implemented) that a symbolic literal of an opaque type name is the only value of that opaque type, and further that that value is unique and unequal to *any other* value whatsoever. The reason that this is a complete head-turning is that instead of opaque types being *compatible with all other types*, they're now *not compatible with any other types*. E.g., the equality check in the following axiom can never type check correctly:

```
object X;
object Y;
axiom (forall x:X, y:Y) x = y;
```

But wait, I just ran this through the type checker an it's fine, i.e., it does not type check.

But wait again (to reject the supposedly "very cool (re?)revelation here" below). Given the statement "... a symbolic literal of an opaque type name is the only value of that opaque type, and further that that value is unique and unequal to *any other* value whatsoever", I believe (seemingly CONTRARY TO the "very cool (re?)revelation" below), that the "`x = y`" equality check in the ***immediately preceding*** example should in fact NOT type check correctly. To be consistent with the values of distinct opaque types being distinct, the types themselves should in fact be distinct. The reason the "very cool (re?)revelation" works in the context of type-var-test.rsl is that there's inheritance going on that's not present in the immediately preceding example. The bottom line is that the immediately preceding example should NOT in fact type check, but the type-var-test example should. The way that the immediately-preceding example could type check correctly is if we liberalized the type checking rules for equality to say that *any* types can be compared for equlity, but that if the types are different, equals automatically returns false. My immediate reaction to this is that it subverts type checking pretty badly, but it's worth thinking through in the context of the "very cool (re?)revelation" that now follows immediately.

OK, I've just had a very cool (re?)revelation here. The revelation is that we shouldn't view the 22mar02 example an *incorrect* result from <, but rather as a *vacuously false* result from <. To recap, here's the type-var-test.rsl ref'd in the 22mar02 injection:

```
obj TypeVar;
obj X < TypeVar = integer and string;
obj Y < TypeVar = string and integer and string;

op CompareXY(x:TypeVar, y:TypeVar) -> boolean = (
    x < y
);

op main(x:X, y:Y) -> boolean = CompareXY(x, y);
```

What we were depressed about here is the fact that given the significant structural difference between `x` and `y`, there's no way we can be doing a meaningful runtime comparison in the expression `x < y`. But hey, that's *just fine* here. What static type checking says in a case like this is that the only thing that can be compared in an op like `CompareXY` is whatever components `x` and `y` have in common, which in this case is *no components at all*, given that they inherit from an opaque type. Hence, in this case, the `x < y` comparison is always *vacuously* false, by the presumed rule that `'TypeVar' < 'TypeVar'` is false. And the reason that `x < y` reduces to `'TypeVar' < 'TypeVar'` in this case is because in the context of `CompareXY` where the inputs are both `TypeVar`, the only thing we can be comparing between `x` and `y` is the common `TypeVar`ness, which means that, as they should be, the other components of `x` and `y` are ignored, i.e., unseen, in the context of `CompareXY`. If the body of `CompareXY` were changed to `x = y`, basically the same thing would happen, except the result would be vacuously true, given in this case we'd be comparing `'TypeVar' = 'TypeVar'`.

So, what this all boils down to is that the only really *wrong* thing in `type-var-test.rsl` is the mnemonic suggestion of the name "`TypeVar`" for the opaque type, because this type name isn't really acting like a first-class type var at all. (And this is in fact what we said in the 22mar02 revelation, but were (hopefully) unnecessary depressed about at the time.) And when all of this gets sorted out, it'll probably be just dandy to have an explicit section in the ref man that addresses this issue head on, perhaps even titling the section something like "Opaque Types are Not Type Variables". Cool.

Now, if we do want real type vars, we're gonna have to add them to the language, which is what this discussion is all about. So, here's what a for-real type var test would look like:

```
obj JustSomeOpaqueType;
obj X < JustSomeOpaqueType = integer and string;
obj Y < JustSomeOpaqueType = string and integer and string;

obj $TypeVar; -- Just for pre-type-var-implementation compilation

op CompareXY(x:$TypeVar, y:$TypeVar) -> boolean = (    -- $ (for emacs)
    x < y
);

op main(x:X, y:Y) -> boolean = CompareXY(x, y);   -- ERROR: actual parameters
                                                  -- are incompatible with type
                                                  -- variable constraints:
                                                  --   exists <($TypeVar,$TypeVar)


obj $TypeVar1; -- Just for pre-type-var-implementation compilation
obj $TypeVar2;

op CompareXYOK(x:$TypeVar1, y:$TypeVar2) -> boolean = (
    x < y;  -- Should be OK once type vars are properly implemented
);

op "<"(x:X, y:Y) = x.integer < y.integer;  -- Should be OK once we get the
                                           -- new/old field ref stuff working
```

```
op main2(x:X, y:Y) -> boolean = (
    CompareXYOK(x, y);        -- Should be OK once we get overloading finished
    CompareXYOK(1, "xyz");    -- ERROR:  actual parameters are incompatible with
                              --         type variable constraints:
                              --             exists <($TypeVar,$TypeVar)
    CompareXYOK(y, x);        -- ERROR:  actual parameters are incompatible with
                              --         type variable constraints:
                              --             exists <($TypeVar,$TypeVar)
);
```

The proposed error message indicates what the problem with `CompareXY` here is. Viz., the body of `CompareXY` requires that there is a "<" operator defined between the two args of `CompareXY`. Further, the signature of `CompareXY` requires that the actuals be of the same type. (Error-messagewise, we may want to tone things down here in the actual implementation, since the type failure in this case is not affected by the lack of a less-than op, but just by the fact that the inputs to `CompareXY` must be the same type. I'm sure we can work out these details just fine. At this point, the error message example is useful to explain the thinking here, and we'll take it as such.)

Now, in the CompareXYOK op, things can work out OK because of the use of two different type vars in the signature and the provided overload of "<". This means that the first call to CompareXYOK works, but the second does not, given that there's no built-in or user-defined overload of "<" for types integer and string. The third doesn't work either, given that the provided overload of "<" is not commutative.

In the Wirthian mode, we should test our new-found inspiration by writing some (more) programs. A particularly good example of where things can come together is in the specs for a Redraw op on graphic canvases, where the power of virtual-function dynamic binding is so evident. A nagging worry about fully static type checking has always been having to do the virtual-binding dispatch logic explicitly by using the '?<' and '.<' operators in a big switch statement that fans out on all subtypes of a parent type. What's particularly worrisome about this is the fact that whenever we add a new subtype, we need to go to all of the places where such dispatch logic exists and add a new case for the newly-defined subtype. This seems like exactly the kind of place where the language should be helping us out. As we know, the problem with the kind of help we get from the O-O crowd is that type checking must go dynamic, which we definitely want to avoid.

Well, I think I may have just come up (and it's probably a reinvention of someone else's smart wheel) with the way to do it with genuine type vars, including quantification over types. The idea is that instead of dynamically dispatching over a subtype, we'll statically dispatch using a type-valued quantification. Here's an example using the Redraw op:

```
object GraphicObject is ...;
object Line < GraphicObject is ...;
object Rectangle < GraphicObject is ...;
object Ellipse < GraphicObject is ...;
object Canvas is graphics:GraphicObject* and ... ;

op Redraw(canvas:Canvas)->canvas':Canvas
    post:
        forall (g in canvas.graphics)
            forall ($GraphicSubtype in GraphicObject)
             if g?<$GraphicSubtype then
                  ProperlyDrawn(g.<$GraphicSubtype)
end
```

Here's the same example with slightly different syntax, which I think I prefer:

```
object GraphicObject is ...;
object Line < GraphicObject is ...;
object Rectangle < GraphicObject is ...;
object Ellipse < GraphicObject is ...;
object Canvas is graphics:GraphicObject* and ... ;

op Redraw(canvas:Canvas)->canvas':Canvas
    post:
        forall (g in canvas.graphics)
```

```
                  forall (?GraphicSubtype in GraphicObject)
                    if g isa ?GraphicSubtype then
                              ProperlyDrawn(g.<?GraphicSubtype)
end
```

Shit, if this is correct, or can be made to be, it's about as scarily invigorating and last night's rediscovery was depressing. What it means is that the type checker is doing the dispatch for us, but it's not at runtime. This is exactly what we're after here. I have a slight feeling that there may be some hidden conceptual problem lurking in here. However, given that we've defined inheritance strictly and completely in terms of unions, it seems pretty clear to me that when we're quantifying over types, it reduces to quantifying over the list-valued semantic representation of a parent type, which is just a list of its inheriting types. I.e., we can see with the formal semantics in mind that this should work out fine.

As a bit of formalization, we can say that the new '`in`' form of the quantifier expression is equivalent to the following more basic form:

```
forall ($GraphicSubtype:object) $GraphicSubtype?<GraphicObject
```

I think a really important observation here is that we're *NOT* going to define a Javaesque top `Object` or a generic `any` type. This means that the keyword "`object`" in the context of the above quantifier expression is not being used as a type name, but as a meta-type indicator that can only be used when the name part of a name/type pair is a type variable name, not a value-variable name. I.e., types are *not* first-class values here (though we might imagine that they could be, meaning we could allow both static and dynamic typing together, but I definitely think we need to keep this can-o-worms closed till we (at least) get the static type var stuff worked out, including, one would hope, an attribute/denotational semantics thereof). What I'm pretty clear we're looking at here is a clarification of our entity categories, that allows us to have value-valued variables (what we have now) as well as type-valued variables, but the twain does not meet between the two, and type-valued variables are only computable at compile time, not at runtime. To ease the lurking-qualm feeling a bit more, I think we're on pretty safe ground here given ML, the only potential problem being the issue of inheritance, which again given the straight-forward union-based semantics thereof should work out. Also, we should read the "adding inheritance to ML" paper that we came across in POPL during the 530 paper scan.

23may02 TODO Item

Change the precedence of `and` and `or` to be less than equality ops, as in CJ. DONE.

23may02 -- Accessing Tuple Fields by Type Name and Ordinal Position

For convenience, the components of an object can be referenced directly by their type names. For example, given the following definition

```
object ABC is A and B and C;
```

access to the components of `ABC` can be made as follows

```
operation Op(abc:ABC)
    pre: abc.A = abc.B and abc.C = ... ;
end;
```

When an object has two or more components of the same type, the components are disambiguated using a positional suffix of the form `#`*n*, as in the following example

```
object PairOfA is A and A;
operation Op(pa:PairOfA)
    pre: pa.A#1 = pa.A#2
end
```

The positional order is from left to right for each component of the same type. In this example, `pa.A#1` refers to the first component in `PairOfA`, `pa.A#2` refers to the second component.

Using the '`#`' disambiguator is the only way to reference compnents by name in objects with multiple same-type components. For example, the reference to `PairOfA.A` without a '`#1`' or `#2` *is an error.*

As another example, consider this definition:

```
object ABCABA is A and B and C and A and B and A.
```

The components of an identifier a of type ABCABA can be referenced as follows:

Reference	Which Component
a.A#1	The first component of type A
a.A#2	The second component of type A
a.A#3	The third component of type A
a.B#1	The first component of type B
a.B#2	The second component of type B
a.C	The only component of type C

.)i where first, second, and third refer to the left-to-right position of components of the same type.

In an object with inherited components of the same type, the positional order is from the top of the inheritance hierarchy downward, and from right-to-left in multiple inheritance order. Consider the following example:

```
object Top is x1:X and x2:X;
object Middle1 inherits from Top is x3:X and x4:X;
object Middle2 is x5:X and x6:X;
object Bottom inherits from Middle1 and Middle2 is x7:X and x8:X;
```

In this case, object Bottom has a total of eight components of type X. The ordinal position of these components is indicated by the names x1 through x8. That is, these names indicate the order in which components are referenced by a disambiguating suffix. For an indentifier b of type Bottom, b.X#*i* references the component that is named x*i*, for *i* between 1 and 8. This example also illustrates that a component can be referenced by either its type name or its explicitly-declared name.

In a tuple with one or more value components, those components can be referecned directly by that literal value. For example

```
object OneAndTwo is 1 and 2;
operation Op(ot:OneAndTwo)
    pre: ot.1 != ot.2;
end;
```

A value component can only be referenced by its specific literal value, not by a variable or constant that contains the value. For example, the following is illegal based on the preceding definition of the tuple OneAndTwo:

```
operation Op(ot:OneAndTwo, i:integer)
    pre: (i = 1) and (ot.i != ot.2);
end;
```

While it may be untypical, literal components of the same value can be referenced using '#' in the same was as other components. For example,

```
object OneAndOne is 1 and 1;
operation Op(oo:OneAndTwo, i:integer)
    pre: oo.1#1 + oo.1#2 = i;
end;
```

Here, since object OneAndOne has two literal components of the same integer value, they can be referenced using '#' to disambiguate them.

There is one additional form of reference for tuple components -- they be referenced directly by ordinal position, without a name at all. For example,

```
object ABC is A and B and C;
operation Op(abc:ABC)
    pre: abc#1 = abc#2 and abc#3 = ... ;
end;
```

A numeric suffix of the form '#*n*' refers to the *nth* component of a tuple, in left-to-right order of component declaration. Hence, abc#1 refers to the A component of ABC, abc#2 to B component, and abc#3 to the C component. ***Probably not:*** The positional suffix '##' refers to the last component of a tuple.

The '#' operator can be used in a sequence of component references in the same way as the '.' operator. Consider this example:

```
object ABC is A and B and C;
object A is D and E;
object E is F and G;
```

For a variable a of type ABC, a#1#2#1 is a reference to the F component of type E.

A *sub-tuple* can be accessed using a range of lower and upper ordinal positions. For example,

```
object ABCDEFG is A and B and C and D and E and F and G;
operation ChangeC(a:ABCDEFG, c:C) =
    a#1..2 + c + a#4..;
```

Operation ChangeC concatenates the first two components of a with a single component c, then with components in the fourth through the last positions of a. The range form #n.. refers to the sub-tuple from position n to the last position.

The use of '#' as a positional component reference is of lower precedence than its use as a component name disambiguator. Consider this example:

```
obj AABC is a1:A and a2:A and b:B and c:C;
obj A is d1:D and d2:D
```

and a variable a of type AABC. In this case, the expression "a.A#2" might be considered ambiguous, since it could refer to a.a2 or to a.a1.d2. The rule used to disambguate such cases is that '#' is interpreted first as a name disambigutor if necessary, second as an absolute position index. Therefore in this example, a.A#2 is interpreted as a reference to a.a2, not to a.a1.d2. In fact, the interpretation of "a.A#2" as a reference to a.a1.d2 is not possible, since "a.A" by itself is an illegally ambiguous reference to one of the two components of type A. To access a.a1.d2 using '#', the legal expression is X.A#1#2.

It is worth noting that use of '#' as a component reference cannot be confused with its use as a length operator. '#' is used as a component referencer when its left operand is a tuple value or a type name. '#' is used a length operator when its right (and only) operand is a list, string, or numeric value.

To summarize the different forms of component reference, consider the following definition:

```
object ABCA is a:A and b:B and C and A and 5;
object A is
```

and an identifier a of type ABCA. There are four forms of reference by which components of a can be accessed:

Type of Reference	**Examples**
explicitly declared component name	a.a, a.b
unique component type name or value	a.B, a.C, a.5
positionally-disambiguated type name	a.A#1, a.A#2
absolute component position	a#1, a#2, a#3, a#4, a#5

The positional ordering of components applies to union objects as well as to tuples. Consider for example

```
object U is i:integer or s:string or b:boolean;
```

and an indentifier u of type U. The type-name and positional references to u's components are defined as follows:

Explicit-Name Reference	**Type-Name Reference**	**Positional Reference**

u.i	u.integer	u#1
u.s	u.string	u#2
u.b	u.boolean	u#3

Given the higher precedence of `and` composition over `or` composition, care must be taken when using positional references in an object defined as a combination of `and` and `or` operators. Consider the following example:

```
object X is integer or string and real or boolean;
```

Given the and/or precedence rules, the default structure of `X` is

```
integer or (string and real) or boolean
```

That is, `X` is a three-elelment union, not a two-element tuple. For an identifier `x` of type `X`, the following table defines the legal type and positional references:

Reference	Meaning
x.integer	the value of x as an integer
x#1	same as x.integer
x#2	the value of x as the two-tuple (string and real)
x#2.string	the string value of the first component of x#2
x#2.real	the real value of the second component of x#2
x.boolean	the value of x as a boolean
x#3	same as x.boolean

Since the right-hand operand of '.' must be an identifier, the second component of `x` is only accessible positionally. That is, there is no such form as

```
x.(string and real)
```

to access the second union componennt of `x`. As explained earlier, such an anonymous component can be given a name, as in the following definition

```
i:integer or sr:(string and real) or b:boolean
```

which allows the second component to be refereneced by explicit name as `x.sr`.

Given the various forms of tuple reference, one might ask under what circumstances the different forms are convenient. *This clearly needs to be finished, with some compelling rationale and examples. We should definitely note that consistency of notation is a very good idea, to make specs understandable.*

Editorial Note: There is definitely a trade-off working here, between notational convenience versus complexity. An argument against convenience goes like this: "Since the main point of SpecL is to be precise and help specs be understandable, it may not be a good idea to have things work in some sense by accident. I.e., being able to refer to tuple components by their type names, which such references can cause understandability problems in a number of subtle ways."

23mar02 -- Limits on (Multiple) Inheritance

There can be no circularities in an inheritance hierarchy. E.g., the following is illegal:

```
object X inherits from Y;
object Y inherits from X;
```

An object cannot inherit more than once from any other object, directly or indirectly. E.g., the following is illegal

```
object X is ...;
object Y inherits from X and X;
```

as is the following

```
object Top is ...;
object Middle is ...;
objct Bottom inherits from Top and Middle;
```

22mar02

See the "(22mar02 injection)" under the 14ded00 entry below.

4dec01 -- Aux Functions as Vaporwear

In thinking about parsing, it seems to me that I might rather not write the aux functions to do this, but instead spec them out. This leads to the concept that one can write aux functions constructively as bodies *or* analytically with pre and post conds. Hmm, does this work? If so, let's make sure we can explain and rationalize it.

And one more observation on parsing -- we need to figure out how the objects-as-BNF-rules scheme fits into things wrt parsing. I.e., can we assume that we automatically have a parser if we write the grammar for something.

The motivation for this comes from Ciera asking about parsing a day/time string into a day/time tuple object. I started writing this up using parsing aux functions in ~/classes/205/examples/specs/parse-day-time.rsl, and started to see how ugly things were getting in terms of reinventing the parsing (or even just lexing) wheel.

4dec01 -- Yet More on Exists as Selector and nil <=> false

OK, in working today with Keiko Tamura on part of the specs for EClass, I had the distinct desire to be able to use exists as an Object-valued operator. If we go back to (or just stick with) the axiom that nil === false, then we could make exists Object-valued. Not-withstanding what's said in ./if-then-else-truth-table and ~/work/rsl/testing/implementation/acceptance/new-inputs/exists-as-search.rsl, I'm pretty comfortable at this point with saying that nil === false.

9nov01 -- More Real Formal Stuff

See the discussion in new-inputs/exists-as-search.rsl

We should define formally that the type of nil is Object. Then we need to read everything we've said about the equivalence of nil and false, and make sure things are consistent. In particular, with remarks to the effect of "I don't know if I'm happy with the notion that (false = nil) = true". Get this ironed out ASAP!

28sep01 -- Type Any, Yet Again

In recent thinking, e.g., retention-and-no-junk.rsl, q.v., we've been considering using the Javaesque identifyer "`Object`" to be the top of the RSL type lattice. While this does have some appeal, particularly in that it helps students and others famiilar with Java relate, I'm at the very moment leaning back to calling the top type "`any`". The reason is that we have a potential conceptual problem with where atomic types fit in the type hierarchy. Java in fact has this same problem, in that the atomic types don't inherit from `Object`. Given these observations, I think it might in fact be misleading to use a capitalized typename as the top of the lattice, rather than a keyword or at least keyword-like identifier such as `any`.

Whatever we call it, there is the seemingly ever-unresolved issue of what operators are applicable to `any`. I think the `retention-and-junk.rsl` provides at least one clearly legal op that can be applied to `any`, viz., `in`, as in `o in out_list`, where `o` is of type `any` and `out_list` is of type `any*`.

So, if we fully enumerate the ops that are applicable to `any`, and explain how it serves as `the` top of the inheritance hiearchy, and in some sense the archetype opaque type, then the ever-unresolvedness will be resolved, and life will be wonderful. You what to do, boy.

27jul01 -- Real-Life Example to Press Understanding of Where 2 Instantiation" 2

The reason the instantiating `where` type needs to be a subtype of what it's replacing is to avoid type-breaking mutation. This is because the where clause is always used in the context of inheritance, which means that if we let an instantiating type arbitrarily change an inherited component, then ref to that component when a value is in a parent-type var could be bogus. E.g.,

```
obj Parent = o1:O1 and o2:O2;
func Foo(p:Parent)->string = p.o1.s;
obj O1 = i:integer and s:string;
obj O2;
obj Child < Parent
  where: O1 = O3;               -- ERROR: this must be prevented per latest
                                -- semantics that say instantiating type must
                                -- be subtype of type it replaces
end;
obj O3 = i:integer and s:O1;
func Foo2(c:Child)->string = c.o1.s;    -- This egregiously breaks typing
                                        -- since if the bogus where clause
                                        -- actually worked c.o1.s is no longer
                                        -- a string, but rather an O1

-- We be in trouble here, since both the preceding and next lines should not
-- be working.  I.e., we can't have it both ways -- c.o1.s is either a string
-- or an O1, but not both.  Somethings wrong somewhere, either because we
-- haven't finished type checking the return type of a function body, or the
-- where clause instantiation is bogus, or some of both.
func Foo3(c:Child)->string = c.o1.s.s;


func Foo4(c:Child)->string = c.o1.s.s.s;  -- Sanity check -- this is in
                                          -- fact an error, as it should be

func Foo5(c:Child)->string = c;         -- OK, it looks like the problem
                                        -- that the type checking of func bods
                                        -- against return types is not done
                                        -- yet.  Never-the-less, the above
                                        -- bogosity observations still hold and
                                        -- need to be fixed.

-- OK, the preceding misses the point that we're trying to get at.  Here it is.
func Foo6(c:Child)->string = Foo(c);    -- Breaks types if where instantiation
                                 -- worked as it appears it should.

-- The deal with Foo6 is that its call to Foo with c as an arg breaks types
-- inside Foo, since c is sent in as a Parent, but when it gets there, its
-- mutated o1.s component ends up mascarading as a string when its really
-- another O1.  This be the reason that where instantiators must be subtypes of
-- what the sub for.
```

The "real-life" place where this has just arrisen is in the discovery that recurring info needs to be different in Meetings versus MeetingRequests in the Cal Tool specs. Hopefully you get the idea well enough in terms of what needs to be done about this, both in the Cal Tool spec and RSL type checker. Get the fuck to it, dickhead.

8jun01 -- Why It's OK to Use Type Name in a Tag Expression

Since or'd components must be distinct, it would be OK to use the type name as the 2nd operand of '?'. This is way-old stuff, but it keeps coming back. The rule could go like this: "If an or'd component has an ident type name, then that type name can be used as the right operand of a '?'".

1mar01 -- Yet More on the Inverted Pyramid Issue

We want/need a way to create a function out of pre/postcond logic. There's mention of this in the book formal-spec chapter, but it's a bit equivocating. The deal is that we need syntax of the form

```
OpX.:post(...)
```

where the ... are all of the inputs (? and outputs?) to OpX. The deal is that the pre and post attributes are defined functions from any to boolean, where the any-valued input signature specializes to arity (?and coarity?) of each op.

The question marks here have to do with not yet understanding how we invoke a function in a postcond and then say that function's postcond is true. In particular, does the function denoted by `OpX.:post(...)` actually run `OpX`, or do we need to run `OpX` first, e.g., in a `let clause`, then call `OpX.:post` with the same inputs as we ran `OpX` with, plus it's returned outputs? Hmm, functionally, it seems at the moment that the former case can't work, given that it would involve some kind of call-by-var semantics we don't have. What we need is a notation that's not dreadfully unwieldy, but still fuctional. THIS NEEDS TO BE WORKED OUT. (Sorry, but I'm not really dropping off here, I just need to get home and wait until a bit later to figure this out.)

8jul02 update: Let's do this for sure. The javaization of the notation should allow "`.pre`" and "`.post`" as suffixes to a class' methods. This syntax is fine, since it doesn't conflict with an existing Java use of '`.`', I'm pretty sure.

16sep02 update: I'm not really sure what I was getting in the paragraphs a couple back, i.e., with the question mark biznis. The deal is, it seems to me, that the functionzied postcond is a boolean-valued function in its own right, to be used in the postcond of some other function as a convenience. So, the call-by-var semantics is not really an issue. I.e., here's what goes on: In op B's postcond we want to say that op A's postcond is true. But in order to say this, B must produce some or all of the same outputs as A. Therefore, the way we use the A.post function is to pass it the outputs of B so that they are validated as they would be for A. That is, what is true of A's outputs is true of B's. The reason, I think, that rasing the call-by-var issue is really a non-issue is functionally, the only way that B can have the same effect as A is for it to produce the same output(s).

But hey, why do we need A.post at all then? Why don't we just equate which ever of B's outputs we care about to the return value(s) of A? While this means that B's postcond "calls" A, why should this be a problem as long as the call is in the context of a boolean expression? Hmm, have we really misunderstood things this badly for so long? I.e., has there never been a need for a functionized postcond at all?? Fuck, this needs to be worked out by examining all of the contexts in which I thought that functionized postconds were necessary, wherever the fuck they are. Per the above note, we should start with the mentioned text chapter.

15dec00 -- Semi-Epiphany

OK, in pondering the 14dec00 TODO LOG entry for typechk.c, I think I'm onto the idea of tagged values at runtime, basically as we've always been onto it, but with an I think new embellishment based on spelling-based type checking. The embellishment is that values with discernable named types will be tagged with that type name, otherwise they're tagged with the structured type.

Well, in writing down the preceding paragraph, I realized that there's nothing really new there at all. Viz., we'll have tagged values at runtime that are tagged with a type struct that can be an ident type or not. Now, what needs to be clarified is exactly when values get tagged with ident types. The answer is under the following circumstances:

a. when bound to a particular op arg of a named type;
b. when built by the built-in constructor for an obj;
c. perhaps other ways, but none that I can think of right now.

The tagged value will let

Now, the other thing to talk about today is why it's OK for '<' to be fully overloaded for all types, including opaque types. The reason is that we'll do structural typing for everything, unless there's an explicit overload of '<' for some type(s) that want it. In this way, it seems that we get the best of both worlds in terms of both full structural comparisons on values that are tagged with unnamed types, versus specialized comparisons on values that are tagged with ident types for which an overload of one or more comparison ops has been defined.

I think that's all there is to it, and so at this point, I'm not quite sure just how epiphanous it was, but it does seem that there were some things said explicitly that have not been so said up to now.

14dec00 -- Nearing Conclusion on 7dec98 BIG BUG

OK, from what I can tell, we *do* want to say the following:

a. forall type T, opaque type OT, T < OT

We've now added this as a (currently the last) type compat rule in

The deal is that since an opaque type has no structure at all, it can have no ops that do anything structural to it, and hence any op that takes any opaque type as an input can have any other type sent to it, since there's nothing the op can do to the opaque type that could be illegal to do to any non-opaque type sent to it.

It would appear that the preceding paragraph effectively answers "Yes" to the question of whether an opaque type is the same thing as a type var. It is the same thing by the very reasoning in the preceding paragraph.

(22mar02 injection -- Look fucker, an opaque type *is not* the same as a type var in the ML sense, since there's no guarantee that two args of the same opaque type will be dynamically the same. See, e.g., new-inputs/type-var-test.rsl. Also, the deal that an opaque type has "no structure at all" and is therefore OK as the parent of any other type is bullshit given structural equality. The deal is that structural equality must know the structure of two types, and given the subtype compat defs we've allowed, all the built-in relational ops must be defined in terms of dynamic types. This is pretty fucking depressing right at the moment, given the type correctness of the "x < y" line in the new type-var-test.rsl file. Again, what this means is that the deal with ML's equlity type stuff is biting us again. I was thinking there may be some way to strengthen static type checking by doing some kind of dynamic binding. Also, I'm questioning right now the idea that we need to say something as strong as X < OT, forall X, OT. Why not just X < OT when that's the way it's explicitly declared? Fuck, this is getting really annoying going around in fucking circles seemingly endlessly here. One final piss-off tonight -- do the spelling rules really mean that X* < Object*, forall X? We need to work this out, since we just added this to typechk.c, where we let the basetypes of two arrays use assmntCompat instead of compat.)

We might ask at this point what good an opaque type is if they're all really the same structurally. Well, the answer has to be in the name itself, and hence there can only be a meaningful distinction between opaque types where the name itself makes a difference. This appears to be precisely the case when we're instantiating a where clause, since we're using the name in the where clause syntax. This seems therefore to obviate the need for any additional '<' or '<=' synatx in a where in addition to the normal '=' syntax. The deal in this regard is that, again, since all types are compat an opaque type, we don't need the '<' or '<=' syntaxes for where clause substitution.

Now, another thot is that the '<' in the above quantifier expression is strictly '<', not '<='. By the current type compat rule, it seems that '<' is strict in the sense that it implies compatibility, not equivalence. This needs (a bit) more thought.

If all this is true, it fixes the big BIG BUG problem, and it should be implemented, as soon as we do the last bit of thinking about it. So, get the fuck to the thinking, and thence implementation.

10dec00 -- Quick Reminder about DD Generator

In ref man revision, don't forget to describe how html tags can be used entity descriptions to improve formatting in DD. Very cool javadoc-like stuff, this.

3dec00 -- Some Quick Thoughts on RSL Manual Updates and Related Topics

In the manual, include an appendix of the current standard libaries.

It's pretty clear that we're going to have an RSL library of common forms. This will require that we finish the implementation of multiple inheritance and the new module import/include forms. Candidates for libraryhood currently include the following:

```
object OrderedCollection, with ops to sort, etc.
function SumList (see new-inputs/sum-list.rsl)
function AddWithRetentionAndNoJunk
object Color
object Text
```

Upon looking at the AddWithRetentionAndNoJunk function, this seems to be like a very nice way to allow reuse of logic without having to go the (multiple) inheritance route. We should describe both ways (i.e., reusiong with inheritance versus aux functions) and be clear about why inheritance sucks. (Or then maybe it doesn't, give most recent musings in sum-list*.rsl about the seeming similarity if not equivalence of the OBJ theory/view concept with the Java interface concept.)

Here's some verbiage in the manual about how to clearly explain the should-be simple concept of enumerations:

> In a number of progamming languages, including C and C++, there is the concept of an *enumeration*. Here is a typical example:
>
> ```
> enum DaysOfTheWeek = {Monday, ..., Friday}
> ```
>
> *[Hmm, maybe we should just not worry about PLs (too much), and just explain enumerations directly in the several rsl forms. OK, here comes.]*

A very typical use of `or` composition is to define what are typically refered to as *enumeration types* in programming languages. Here is an example:

```
enum DaysOfTheWeek = {Monday, ..., Friday}
```

From here we describe that `Monday`*, etc. are the enum list, and then describe how to represent them as either strings or symbolic literals.*

Here's some verbiage for the ref man about the difference between strings and symbolic literals:

> The difference between double-quoted strings versus single-quoted symbolic literals is subtle but significant. First off, if you don't want to worry about the difference, you can get along just fine in RSL by using strings everywhere that you might otherwise use a symbolic literal, and just forget about symbolic literals entirely. If you care, read on.
>
> *Now go on to describe how symbolic lits are more abstract; that the string/symbolic-lit distinction is precisely the same as the string/symbolic-atomic distinction in Lisp; that when modelling views or other objects that have abstractly atomic but concretely non-atomic structure, symbolic lits are a more accurate model (albeit subtly so); also when that when modeling grammars, symbolic literals provide the means to define abstract versus concrete syntax.*
>
> And fuck me if that doesn't all make some pretty good sense.

3dec00 -- Some Details on the New import/include/export Scheme

To get the effect of nested modules, we do like this

```
module Outer;
  export *;
  include Inner1, Inner2, Inner3;
end Outer;

module Inner1;
  export X;
  obj X;
end Inner1
```

As in Java, we don't physically next module inside of each other, but create a form of "virtual" nesting via include. Now, it seems to me that the only reason we need nesting is for namespace control, which Java does by qualifying the name of the package that a file's worth of stuff is in. For us, we get the same effect by a definition of the Outer form above, since we can say

```
module User;
  import Outer;

  op Use(x:Outer.Inner1.X) = ...;
```

The point here is that if we assume that the names "Inner1" and "X" are both potentially widely-used symbols, then we want not qualification of "X" but of "Inner1" as well. If we didn't care about this, then we'd write User like this:

```
module User;
  import Inner1, Inner2, Inner3;

  op Use(x:Inner1.X) = ...;
```

We should note that export makes available not only the entities directly listed, but the transitive closure of the components of the entities for those who use the '*' form of importation.

In summary, we have the following consistent and orthognal semantics:

a. "import" and "include" both make exported symbols visible in module.
b. The (only) difference between the two is is that "import" is qualified and "include" is unqualified.
c. The '*' form means do a transitive closure on all components. We need to be careful and fully clear here about what happens when we transitively close through a series of nested modules. It shouldn't be a problem, we just have to make things completely clear. We may want to go back to the Mod-2 definitions where this was spelled out, including what I recall was a difference between version 1 and version 3, or something like that, as well as the idea of import/export of a record working transitively on the record fields.

30nov00 -- More Ideas on Executability

Problems and issues include:

- Implementing unbounded quantification by quantifying over all allocated object values, where allocation takes place by invocation of a constructive op. The problem here is that if we execute via only validation invocation, we don't really construct objects. Obviously, we need to think about this. Oh, I guess I have thought about it -- it's down in the next section under I had a flash today
- How to show, or distinguish, validation invocation in a DFD. A thought is to specify an overall mode for DFD (and postcond) invocation, rather than a special form as shown below. However, such global modes don't seem particularly appealing. Anyway, we need to think about this some more.

14nov00 -- Ideas on Executability

One form can be providing input/output pairs and having the pre and postconds evaluated. I'd like to have first-class RSL invocation syntax for this form of "validation invocation", which might look something like this:

```
op Foo(i:integer, s:string)->(i':integer, s':string)
    pre: i > 0;
    post: (i' = 10) and (s' = "abc");

op Test() =
begin
    Foo(1,"xyz")?->(10,"abc");    (* Produces value {true, true} *)
    Foo(1,"xyz")?->(9,"abc");     (* Produces value {true, false} *)
    Foo(0,"xyz")?->(10,"abc");    (* Produces value {false, nil} *)
end
```

The idea is that the invocation form

`op-name(value, ...)?->(value, ...)`

always produces a tuple of type `boolean and boolean`. This looks awefully sweet to me at this point.

Note that when the precond evals to false, the postcond *must* eval to nil.

Dealing with quantifier eval will of course be an issue. I had a flash today that we could deal with unbounded quantifiers by saying we quantifier over all the concrete objects of a given type that have been instantiated over the course of a test, where instantiation means that an object has been created and bound to a currently live location (i.e., in a live var directly or within a composite value that's bound to a live var). In this way, if we wanted to have a large pool of active values of a particular type over which to quantify, where the pool holds what we consider to be a *representative* collection of values, we could do something like this:

```
obj IntAndString is integer and string;
op GenInstancesOfIntAndString(i:integer,s:string)->IntAndString* = (
    if i>10000
    then []
    else [i, string(i)] + GenInstancesOfIntAndString(i+1, "")
);

op Test() =
begin
    var int_and_string_pool:IntAndString*;

    ...
```

Another way to do things might be to quantify over all values of a given type *ever created*. In this way, we wouldn't need to stick the values in a list, though we could if we wanted. This is sort of MLish, in the sense that values live forever. And I think we have a firm handle on "created" -- value creation happens whenever an op that outputs a particular type of object is successfully executed.

This hints at the idea that we could define some sort of *quantifier-generator* function that defines over which specific values of a given type we quantify. We could define such a function as a set of constraints over all values within a given type. This sounds interesting, and definitely worthy of further thought.

19oct00 -- Making it up as we go along, some more

Hmm, it just occured to me that we may want to make quantification of types use name equivalence. We need to think (probably hard) about this.

19oct00 -- Making it up as we go along

Well, we've now decided, I hope correctly, that a forall with a failing quantification clause, e.g. forall (x in []), returns true. The rationale is, threefold. First, it seems by far more convenient and intuitive than returning false. Also, it's consistent with the truth table for implication, which returns true for true => false. Finally, if we look closely at the supposed redcution translation of the "in" form of forall, the implication truth table business seems to be supported.

16nov01 update. Here's the "look closely" part in the last sentence of the previous paragraph. Consider that

```
forall (x:O | p1(x)) p2(x) <=> forall (x:O) if p1(x) then p2(x)
```

Now, if p1(x) is never true, this means that "if p1(x) then p2(x)" is always true, which means that the quantifier succeeds. What this means intuitively is that if the quantifier variable clause yields no values at all, the forall quantifier is (vacuosly) true. I.e., anything is true about nothing. This last statement has the same counterintuitive feel that logical implication has, viz., false implies true. Be that as it may, things do indeed work out here the way they're formulated, where "work out" means being sound wrt standard predicate logic, most particularly the truth table for implication, as funky as we sometimes think it is.

Complete discussion of this needs to go into the ref man.

19oct00 -- Making if-then-else formal

Though I don't think I've openly acknowledged it excplcitly before, the problem with if-then-else is when it is used as something other than a bool-valued function. We really do need to clear this up formally. I.e., is if-then really equiv to implies, or do we need to define it in Lisp logic terms?

As of 15nov01, the answer to this is in ./if-the-else-truth-table.

19oct00 -- Clarifying Condition Inheritance in Ops

I actually don't remember the exact rules for op inheritance in the O-O sense. It occurred to me that it might be a very nice feature to say that an overloaded op only inherits pre- and postconds where the specifier explicitly requests such inheritance. We could do this by constraining the conditions underwhich *effective* ops are generated via inheritance. Viz., the effective op generation takes place only for ops that are listed explicitly in the operations clause of a parent class. In this way, one can allow logic not to be inherited, in the case that an effective op didn't want the inheritance.

We need a good example of this. The cruxt of what we're saying here is that an inheritance-generated effective op would not want some or all of the input constraints that were defined for the parent-object op. Again, I can't think of a good example right now, but it is worth investigating, and I think worth including as a feature to allow more specification flexibilty.

I'm not entirely sure about the last point (i.e., that flexibility is always necessarily a good thing), so the issue definitely requires more thought.

21sep00 -- Dumping Coarity-Only Overloading

Per the 7jul00 LOG entry, it's time to dump coarity-only overloading, for the reasons listed there. Implementationwise, there's not much to dump, since we're far from a complete implementation. What we need to get rid of in the implementation are the current quick hack that allows co-arity overloading for a very specific case, plus all the hooks that were put in to support the future full implementation of coarity-only overloading.

Conceptually, the decision is now final. Viz., there is no coarity-only overloading in RSL.

5jul00 -- Resurecting Formal Dataflow Defs

This is pretty bizzare, but I thought this was written down. Anyway, here's the idea. Oh fuck, I just found it (the example I was thinking about) in se-book/semi-formal-spec. Anyway, again, here's some more pertinent discussion.

We probably want to disallow anthing but AND'd op components. If not, then we need to specify the semantics of OR and STAR'd components. There has been some thinking (in the pretty dark past) about defining these as non-determinancy and unfolded loops, respectively. However, with OR we'd have to reconcile with the connections to make sense of it. Similarly, it would seem, for STARs. Anyway, I think we'll do best just to disallow anything but AND's and say that some future version of RSL may lift this restriction. One argument against never allowing ORs is that non-determinism sucks pretty much (at least for my meager brain at this point, and probably forever).

2jun00

In the new semantics, we need to be clear in the ref man about the following:

```
obj TheStringXYZ is "xyz";
val TheValueXYZ  = "xyz";
val TheStringABC:TheStringXYZ = "abc";  -- ERROR: the LHS type is
                                        --   "the string xyz"
                                        -- but the RHS type is
                                        --   "the string abc", which aren't
                                        -- compat by the normal type rules
```

OK, forget about it in just the ref man, we need to be clear about it period. Question: "What's the type of `TheStringXYZ` and what is it compat with?". Ans: The type is "the string xyz" and as a type its compat (but not equiv) with string. The tricky bit we're getting at here is the following: A value-constrainted type cannot have unconstrained values assigned to it. I.e., an ident of type `TheStringXYZ` cannot have any string other than `"xyz"` assigned to it.

This value-constrained type-definition rule does not apply to value definitions (well now it does -- see "But wait" below). I.e., when a value identifier I is assigned a literal L that denotes a specific value of some type T, I is defined to be of type

T, not `(the T V)` (actually not -- see again the "But wait" below). E.g., `TheValueXYZ` defined just above is of type `string`, not `the string xyz`. But since values are constants, they can never by rebound, so there's no issue of what it would mean to (re)assign some value other than "`xyz`" to `TheValueXYZ`. This effectively means that `TheValueXYZ` acts just like a variable of type `TheStringXYZ`, even though technically they're of differenct types. The point is by compat rules, `TheStringXYZ` is compat (but not equiv) with `string`, and `TheValueXYZ` *is* is equiv to string, but since it's a constant it cannot be rebound.

But wait! I think we want to fix the small anomoly in the preceding paragraph by saying that a value *is in fact* of a value-constrained type. Since value-constrined types are upward compat with their "parent" types, this won't put any damper on how value can be used. E.g., `TheValueXYZ` is in fact of type `the string xyz`, not of the more general type `string`. This does not constrain how `TheValueXYZ` can be used, and seems to allow for a more straightforward implementation where all constant literals are formally defined (and implemented) to be of value-constrained types, which means that the implementation of a value declaration just copies the value-constrained type of the constant into the type of of the constant ident.

Now, what we don't have to worry about now that obj's and val's are distinct, is that TheStringXYZ is a value -- it's not. Therefore it cannot be bound to any ident, which means we don't have to worry about what it's compat with in any binding context. Well, this isn't exactly (if at all) right, but there's still not a problem. Viz., an ident of type TheStringXYZ can be bound to an ident of type string, per the normal type rules. I.e., TheStringXYZ is compat with string. So fine.

Now here's an interesting question -- does a var of a purely value-restricted type ever need to be, or can it even be, bound? E.g., what's up with a var decl of the following form?

```
var the_string_xyz:TheStringXYZ;
```

We might say that since it can only hold one (non-nil) value, that that value is automatically bound to it when it's declared. What I don't like about this is that it makes an exception to the rule that there's no auto-binding (except perhaps to nil) of vars. What this means is that the_string_xyz could in fact have the value nil, until it's bound to "xyz". And the reason this seems a bit funky is that we have to assign a value explicitly.

And here's (perhaps) a bit more funkiness. When we declare a tuple field to be a specific value, should that tuple field as a var be automatically bound to that value? I think the confusion I'm verging on can be resolved with the following example:

```
obj AnIntAndAnXYZ is i:integer and the_string_xyz:"xyz";
obj AnIntAndAnXYZ is i:integer and the_string_xyz:TheStringXYZ;
```

Hmm, maybe this example can't quite resolve things (later note: but read on, because it can). The question is are these two types equiv? It looks like the spelling rules say yes. In either case, the deal is that we want a value of this tuple type to have constant value in the second element. But if we declare an ident of this tuple type, it has to be explicitly bound just like any ident, with the restriction that the second field can only be bound to the value "xyz". So it looks like there isn't any diff between the var and tuple case. Either way, a type with one or more value-constrained components is still just a type, not a value. If it's a degenerate case of all pure-valued components, so be it. The fact that we call it "degenerate" means that its behavior can be (a bit) funky. The funkiness in this case is that it still needs to be explicitly bound to a value, even though there's only one non-nil value it can possibly have.

Now, it appears that a union op turns a pure-valued type into a variable type, in the sense that more than one value can be bound to it. E.g., compare

```
obj OneAndTwo is 1 and 2;
obj OneOrTwo is 1 or 2;
```

The first can only be bound to one non-nil value, whereas the second can be bound to two. I'm not sure if we need to say anything more about this, other than to be clear that union types create a form of value multiplicity (even) when the union components are pure values.

Now, whither the concept of "constant"? Well, I think we can say definitively that a `value` is synonymous with "constant". But as the preceding discussion (hopefully) makes clear, an object (aka, type) is not (can never be) a constant, since an object (aka, type) is never a value.

Now for the ref man explanation. When an obj definition is a specific value, we've created a type of object that can only hold the single specific value that it is defined to be. On the other hand, when we create a value of a specific literal-denoted type, the literal in general denotes a broader type than its specific value. It's only when that type is bound to a value-restricted ident that the type becomes less general. Some further examples here would almost certainly be in order.

One of the examples can make completely clear why the ERROR comment above is true. E.g., the value "abc" can be bound to a string, because by type rule (b) "the string abc", which is the type of the value "abc", is compat with "string". However, there is no rule that makes "the string abc" compat with "the string xyz". QED.

And finally, let's probably give a ref man example such as the following

```
obj ABC_Type is "abc";
var abc:ABC_Type := "abc";  -- holds the value "abc"

val ABC = "abc";            -- denotes the value "abc"
val strictly_ABC:ABC_Type = "abc"  -- denotes "abc", but in a stupid way
```

which illustrates that declaring an object to be of a single value is just a round-about way of declaring a constant value, where the "round about" part is that one must also declare and assign a variable of that type in order to have a usable value on hand. The only (rather useless) distinction between the variable `abc` and the value `ABC` is that the variable can hold the value `nil` but the constant will never denote `nil`.

And finally finally, we should probably have an example that illustrates that a fully contant tuple type isn't any more useful than value-constrained scalar type, in that it *can* only hold values of one type for all of its components, but that as a type it *does not* "contain" values at all, and therefore a variable (or constant) of the type must be declared.

Yet Further Analysis Related to 19oct99 Note (31may00)

This item, i.e., all of the discussion under this 31may00 heading, appears to draw a pretty definitive conclusion about subtyping and compatibility. As we'll see, the discussion here has led to an update of the spelling-based formal type rules, and succeeds (I'm pretty sure correctly) in demystifying the lingering unclarity about subtyping versus subsetting.

So, let's start by reconsidering the example presented in the "More Analysis" item just below:

```
obj OneOrTwo is 1 or 2;
op main(oot:OneOrTwo, i:integer) = begin
    oot = 1;  (* Fine *)
    oot = 2;  (* Fine also *)
    oot = 3;  (* Clearly not fine *)
    i = oot;  (* NOT fine *)
    oot = i;  (* Not fine, perhaps not so clearly *)
end;
```

The change here is that the assignment "`i = oot`" is now considered NOT fine, because we can reason by the (pre-31may00) spelling-based formal type rules that it doesn't work. Here's the reasoning:

a. The type spelling of *`oot`* is "union of (the integer 1), (the integer 2)".

b. The type spelling of *`i`* is "integer".

c. By application of the spelling-based compat rules, "union of (the integer 1), (the integer 2)" is not compat with "integer". What the potentially applicable rules do say is:

   i. "the integer 1" is compat with integer (by compat rule b)

   ii. "the integer 2" is compat with integer (again by compat rule b)

   iii. "the integer 1" is compat with OneOrTwo (by compat rule c)

   iv. "the integer 2" is compat with OneOrTwo (again by compat rule c)

d. I.e., we can stick a "the integer X" into an integer or into a OneOrTwo. But just because we can do this does not mean that we can stick a OneOrTwo into an integer, or vice versa.

Now, here's an interesting slip I originally made at point c in the above reasoning:

...

a. This means that `oot` is *compat* with `i`, but *not equiv*.

I.e., a value of the type spelled "the integer *X*", for any *X*, *can* be bound to an identifier of the type spelled "integer".

i. However, a value of the type spelled "integer" *cannot* be bound to an identifier of the type spelled "the integer *X*", for any *X*.

This reasoning is stupid and in fact totally going around in circles (based on the pre-31may00 type rules, which are about to be amended). It's stupid because the subpoints i and ii don't really support the superpoint c. In order for this to happen, one of the subpoints would have to say ‘‘A value spelled "union of (the integer 1), (the integer2)" is compat with "integer"’’, which is not true by the current (pre-31may00) formal type rules. The reasoning is going around in circles since it's the (kind of) reasoning that was (apparently) used to think originally (i.e., in and around 19oct99) that "`i = oot`" was OK. (Hmm, or even fuck!)

Anyway, let's investigate what it would take to get OneOrTwo to be compat with integer. It looks like we'd have to have a (new) type rule of the form:

a union type is compat with type X if all components of the union type are compat with X

What's going on here is that we're considering a union type in which all elements have a common compat-parent, but none is directly compat with each other. This appears to be essentially the subtyping rule for compatibility. I.e., each element of common-compat-parent union acts like subtypes in the sense that none is compat with each other but they all share a parent with which they are compat. (This is just rethinking of the basic oo rule that subtypes are compat with parent types).

Now, another way to look at things is that what we've got going here are two ways for a type to be compat with another: (1) via being an element of union, which makes the element type compat with the union type, which by the union-based definition of inheritance is the same as subtyping; (2) via being a type of the form "the X Y", which makes the "the X Y" type compat with X (which I guess can be called the "subsetting" rule (and see the conclusion of this note item below).

Next consider that it's silly to have an explicitly-declared union of types that have a common inherited parent, because that union type would be completely redundant with the automatically-generated inherited parent type. (Recall again that with the union-based definition of inheritance, a parent type is in fact a union of its subtypes.)

Consider, however, that unlike with subtypes that are explicitly declared, there is no explicit set of (infinite) declarations that say a type X is defined as the (infinite) set of all types of the the form "the X Y". So, it does make sense to have an explicitly-declared union of value-style (i.e, of the form "the X Y") types that share a common compat parent. And, since there are only two ways to be compat (as opposed to equiv), the kind of union we're talking about must be value-style. And it looks like what we've uncovered here is another way to define a collection of types with a common parent. Viz., the set of all types of the form "the X Y" have the common (compat-)parent "X". Given this, it seems reaonable to add the above new compat rule, since it provides a measure of uniformity for things. What's a little funky is the idea that it's silly to have a union of types that all inherit from the same type, but that's what we get for having the automatic rule that maps inheritance to subtyping.

To make everything crystal clear here, I think we must also consider the relationship between automtically-defined parent-class union types and the new type rule. An automatically-defined parent-class union type is of the form

"union of (tuple of *original components*), (tuple of *original components*, *child 1 components*), ..., (tuple of *original components*, *child n components*)".

Now, by the new type rule, what is this auto-gen'd type comapt with, if anything? I.e., does there exist a type *T* with which each of the auto-gen'd components is compat? Well, except in the degenerate case when all child types are exactly the same type, it appears that the answer is "no". This is because each component type is at type-level an explicit tuple, so the new rule cannot be (recursively) applied to any of the component types to get a compatible type. And unless each child type is the same, there will exist some component of one of the tuples that is distinct from the others, hence precluding the exisitence of a common parent-compat type *T*.

So, now that we've reasoned that such a new type rule would be OK, the question is do we want to bother with it? I.e., does it really buy us anything useful? Well, by our reasoning, the only kind of union types we're dealing with to which the new rule would usefully apply are those with value-style components. For these types, is it a big deal (or even much of a deal at all) to be able to do the kind of bindings that the new rule would allow? Consider again the example we've been working with

```
obj OneOrTwo is 1 or 2;
op main(oot:OneOrTwo, i:integer) = begin
    (* ... *)
    i = oot;  (* NOT fine *)
    (* ... *)
end;
```

Is it useful to make this type of assignment (and type-comparable parm bindings) legal? Well, if we don't make them legal, what would we have to do to make them happen? I.e., how (if at all) can we assign a value of the OneOrTwo to a var of type integer? Well, it looks like we cannot do such a whole-variable assignment, but rather must do explicit projection of the component elements. And to do this in the current structure of the language, it looks like we must give each component a name. So, e.g., what we're trying to do would look like this:

```
obj OneOrTwo is one:1 or two:2;
op main(oot:OneOrTwo, i:integer) = begin
    (* ... *)
    i = oot.one;
    i = oot.two;
    if oot?one then i = 1 else i = 2;
    (* ... *)
end;
```

But this looks pretty brain damaged conveniencewise. However, what's all this accomplishing any way? I.e., why do we want to assign a restricted integer value to an unrestricted integer? It's akin to, if not essentially the same as, being able to assign an enumeration literal to an integer, which in a nicely strongly typed language need or ought be allowed. The point is the following. By creating a type to be one of a restricted set of of constant values, why not just require that we deal with it in a little island of restricted value manipulation that includes only explicit references to the constant values?

Well, a good answer to the last question is not in binding, but in conditional logic. E.g., consider the following example:

```
obj OneOrTwo is 1 or 2;
op main(oot:OneOrTwo, i:integer) = begin
    i = SomeIntFunction(...);

    if (i  = oot) then
       (* ... *)
    else
       (* ... *);
end;
```

Without the new rule, it seems we'd have to do the following:

```
obj OneOrTwo is 1 or 2;
op main(oot:OneOrTwo, i:integer) = begin
    i = SomeIntFunction(...);

    if ((i = 1) and (oot = 1)) or ((i = 2) and (oot = 2)) then
       (* ... *)
    else if (
       (* ... *);
end;
```

which looks pretty painful.

So, I've done it. I've added the new rule in the formal-type-rules file. It seems a bit hokey, given the power of the other rules versus the relative special-caseness of the new rule, but I think we can live with it. To mitigate things in this regard a bit, the only reason that compat rule c is all that powerful is because of the unionization of subtyping, without which we'd have some more rules about subtype compat. Anyway, it's done (at least for now) and let's live with it (at least for now). Don't forget that we still have to implement all of the formal type rules!!

So now, I think we can *at last, once and for FUCKING ALL*, defuckingmystify the issue of subtyping not being subsetting. The demystification, it appears, has been in the spelling-based formal type rules all along. Here's the deal:

a. We have the following two *distinct* type compat rules:
   i. a type spelled "the X Y" is compat with X;
   ii. the type X is compat with any union type of which X is a component;
b. Rule *i* addresses the subsetting part of the issue whereas rule *ii* address the sutyping part of the issue. The reason that *ii addresses subtyping is, again, due to the unionization of inheritance.*
c. *The deal is, subtyping and subsetting are not the same thing, but by these rules they do individually imply the same thing.*
d. *In strctly logical terms:*
   i. let st be the propsition subtyping
   ii. let ss be the propsition subsetting
   iii. let c be the propsition compatibility
   iv. The compat rules say effectively the following:

   (st => c) and (ss => c)

   v. But this logic certainly does not imply that st <=> ss, or even a one-way implication between st and ss.

The point is that we've apparently been suffering in some anti-logical fog about all of this, thinking that there's some kind of relationship between *ss* and *st*, but not being completely (if even nearly) clear on what it is. Well, it now appears clear the the relationship is that they both imply compatibility, which in logical terms leaves them otherwise unrelated. Cool (I think and I hope).

The "More Analysis" Called for at the End of 19oct99 Note

One of the things we need to get straight in terms of type safety is that a type with specific enumerated values of, say, type int, is *NOT* compat with type int. E.g.,

```
obj OneOrTwo is 1 or 2;
op main(oot:OneOrTwo, i:integer) = begin
    oot = 1;  (* Fine *)
    oot = 2;  (* Fine also *)
    oot = 3;  (* Clearly not fine *)
    i = oot;  (* Fine *)
    oot = i;  (* Not fine, perhaps not so clearly *)
end;
```

What we have going here appears to be inheritance-oriented subtyping in reverse, at least in terms of the cardinality of the types. E.g., when T1 < T2, #T1 > #T2.). What's curious about this, which in fact has been a very long-standing curiosity, is that subtype is *IS NOT* sub*setting*, since the cardinalities are reversed.

Here We Go Again (19oct99)

(3dec00 addendum to the addendum -- I already know I'm a stupid dickhead. See the 31may00 item above.)

(3dec00 addendum -- This was already figured out earlier, stupid dickhead. So, the "I'm not sure" heading is irrelevant. Viz., it *is* the case that the latest spelling-based semantics allow values to be part of types.)

I'm not sure if the latest semantics (defined below and in be defined as a specific set of values. This will let us get back to the old way of doing enum literals like

```
object Sex = Male or Female;
value Male = "Male";
value Female = "Female";
```

What seems to have been considered problematic before was to have this definition of type Sex be "string or string". As observed below, this isn't a good idea, since a union of the same types isn't all that sensible. But really, Sex should not be considered of type "string or string", but rather of type "value "Male" or value "Female"", which may or may not be a subtype of string or string, but is decidedly not equal to type string or string.

What still remains problematic is a syntax that would allow unnamed constants to appear directly in type defs, as in the following version of the above

```
object Sex = "Male" or "Female";
```

I believe the seat of the syntactic problem here is the redundancy of the "and" and "or" operators in comp exprs and value exprs. Hence, we can't have the fully general syntax we'd like of allowing any value expr as a component of a type. However, we can have numeric and string literals as syntactically allowable components, which renders the immediately preceding definition syntactically legal. (This syntax has been tested in a version of the parser, but is currently commented out since it causes type checking problems. See newtests/string-and-num-lits-as-components.rsl.)

There is a small potential problem of having an ident designate both a type and a value. Probably the easiest and most sensible way to avoid this problem is to simply disallow it. I.e., an ident cannot designate both a type and a value in the same scope. This may be inconsistent with some part of the current wanna-be formal type semantics, but I don't think it's a fundamental problem at all. I.e., the current semantics can be easily reconciled with the idea that type and value name spaces must be disjoint in a given scope.

What's nice about this thinking is that we've come full circle to some extent, in that we'll now allow the old notion of strings as enum literals. What may also be nice is to do away with symbolic literals entirely, if possible, since they really are excess baggage to a certain (if not large) extent. The old thinking in this regard was always to have strings be a single concept of actual string data as well as enumeration literals. We may be able to get back to this original RSL concept.

And here's one last little nugget in this topic area. Question: should a type like "value "Male" or value "Female"" be (automatically, definitionally, formally) a subtype of "string"? If so, by what formal rule? Similarly, should the type "value 1 and value 2" be a subtype of "int and int"? Again, by what rule.

An initial crack at a rule like this might go something like this:

a. an or'd value set of values of the same type is a subtype of that type

b. an or'd value of a set of values of different types is a subtype of an or of those types

c. an and'd value set is a subtype of an and of the value types

This does require some more formal analysis to make sure it really works.

(And another 3dec00 Metanote: you're a still a double (if not triple) stupid fucking dick head, because the 31may00 item above totally says, and better so, what's in the next 3dec00 parenthetical remark.)

(Another 3dec00 Note: I think you're a stupid dickhead. I'm pretty sure that a careful reading of the (long) extant spelling based compat rules naturally allow what we're gettin at here. Specifically, the following compat rules

- a type spelled "the X Y" is compat with X (but very importantly, *not vice versa)*

- *a type X is compat with any union type of which X is a component (but again, pretty obviously, not vice versa)*

handle the (exactly) the case we're talking about here, given that the spelling rule type "the X Y" is the same thing as we're above as "value X".

I'm pretty sure that the conclusion here is "wake the fuck up, the spelling-based type rules really are what we want". So, among other things, stop calling things "subtypes" when you shouldn't be.)

More Syntactic Cleanup

In addition to the stuff about `value` keyword ... damn my eyes -- I fell off here again.

Answer to the 7dec98 'BIG BUG' LOG Entry

The observation about a major conceptual bug in the 7dec98 LOG entry is evidently correct. The immediate answer, after some thought, is that rather than require that an instantiating type be a subtype of the generic type, we should require that all generic types be opaque. This makes sense from at least one major standpoint, viz., the basic structure of the `where` clause. When we say "`X = Y`" in the `where` clause, the intuitive interpretation is that of full substitution. In this sense, saying that Y must be subtype of X, the more sensible where clause notation would be "`X > Y`", which (perhaps flash), we might actually add. But let's discuss some more basics first.

The reason that requiring all generic types to be opaque makes sense is that when we do the where instantiation, we are intuitively substituting one type, `in total`, for another. Hence, that a generic type has any component properties at all seems stupid from this perspective, since any such properties are completely wiped out by the instantiation.

Now, we have apparently had in mind that instantiation is not really of the "wipe out" variety, but more of the "specialization" variety, wherein any properties that the generic type has are *specialized* by rather than replaced by the instantiating type. And this seems to make some sense, though this may be the first time that we've actually written it down in this way.

Now, what may be possible, either notationally, or via further clarification/extension of opaque type semantics, is to have it *both* ways. Viz., we will effectively, if not explicitly say the following: "If a generic type is opaque, then it can be instantiated with *any* type; if a generic type is not opaque, then the instantiating type must be a subtype of the generic type".

Now here comes the potentially cool part. Suppose we say that any type *is* in fact a subtype of any opaque type. In this way, we wouldn't have to state the instantiation rule conditionally (!).

So now what we need to do is to investigate if the rule that all types are (automatically) subtypes of an opaque type makes sense. We'll do this in the next item or two.

Type Definition and Subtyping at the Margins

OK, en route to our goal of all types being subtypes of opaque types, let's do a little lemma-like reasoning, probably again, on the nature of opaque types. If we've not said so already, it seems reasonable to say that an opaque type is a zero-tuple. (It turns out that grepping for {0,zero}-tuple turns it up only once in typechk.c, where the comment is to revert objs that turn out to be 0-tuples back to opaque types. This comment is apparently consistent with what we're saying now, viz., that we formally regard opaque types as 0-tuples.)

Anyway, this being the case, here's an interesting postulate:

```
obj X is T
```

and

```
obj X < T
```

are the same definition. In the first case, X is a one-tuple of T. In the second case, X is a zero-tuple that inherits from T, which turns it into a one-tuple of T. OK, so this really isn't too startling.

So, what we're saying here is that at the margins, a one-tuple is-a opaque obj that inherits from one type that is-a identity for the defining/inherited-from type. And I don't think this is new at all in the fundamental semantics as currently defined in the typechecker, documentation, and our heads.

Back to the Latest 'Fundamental' Question

So, are we OK to say that an any opaque type is automatically a parent type of any (non-)opaque type? Here are some more questions in this regard that we'll need to answer:

a. Do we need the "(non-)" just above; i.e., can opaque type be a subtype of another opaque type, including hence itself? I'm leaning "no" initially on this.

b. Should we soften this idea to say that an opaque type is automatically *allowed* to be the parent type of any non-opaque type? In this way we're talking about the potentiality of being a parent type, which may possibly be better (if it is in fact any different formally).

c. Should we add a "<" (or, per the change in syntax recommended below, "<=") form to the where clause, that explicitly specifies inheriting rather than replacing instantiation semantics. It seems that this would be cumbersome and potentially confusing notationally.

d. Will this rule for opaque types mix OK with the probably-not-yet-fully-articulated idea that an opaque type is a form of type variable?

e. Oh, and one last somewhat related, and I think still nagging question: does it make any sense to inherit from an atomic type?

I'm pretty happy for now to stop here, think about it some more, and come back to finish things up. I'll be back.

OK, I'm back now, with the following thoughts (and hopefully a conclusion).

As usual, I was the total fucking that I am, and did not conclude things here. The latest thinking, which is pretty close to a conclusion, is in the 14dec00 entry way above.

Possible Syntactic Superflash

OK, if we go ahead as stated below and put in `value` as a keyword, the "obj =" form of definition is now deprecated. How about if we now use "=" in place of "is" and loose "is" ***altogether*** from the syntax. The benefit is that we won't be dealing with the loaded English word "is" anymore, which has all the "is-a" inheritance baggage.

So, what we'll be left with are "=" and "<" for the two shorthand definition forms, or we might even want to use "<=" instead of "=". So, we'll have

```
obj X = A and B and C
```

and

```
obj X < Z = A and B and C
```

Some more examples, to feel things out:

```
obj PersonRecord = Name and ID and Age;
obj SalariedEmployee < PersonRecord = Salary and Step;
```

Having both short forms together does look a little hard to follow, but we can say that the user can always go to long form. We might want to throw in a semi-colon, as in

```
obj X < Z; = A and B and C
obj SalariedEmployee < PersonRecord; = Salary and Step;
```

but I'm not sure this helps much. Bottom line, I think the long-form notation is plenty fine to alleviate the possibly confusing look of using both "<" and "=" in the same short form definition.

An initial reaction here is that the only (probably slight) drawbacks are we loose the someone nice-feeling "is" for the most early introduction to formal specs, and we change the semantics of "=" compared to earlier versions of RSL. The first I expect we can live with just file, by saying "Hey, look, we're starting to get formal here, so we're OK using the slightly more 'formal looking' is A and B". Plus, the major win here is we loose the potentially misleading use of "is" as what amounts to "has". What we're saying is that we'll loose both "is" and "has" entirely, to avoid altogether the whole confusing pseudo-English definitional mess.

The second drawback is a bit more serious, but hey, probably not that many people (e.g., mostly Dan) ever used the "=" form of object defs anyway, so we'll be OK. Plus we'll write him a nice Emacs script to go through and change all of his RSL examples. Plus he'll probably agree with the idea of eliminating the confusion around the (mis)use of "is" in RSL. Another mitigating factor regarding the semantic change of the older "obj =" definition form is that we seem to have committed to the value keyword syntax, which totally replaces the old meaning "obj =" anyway. So it appears that we in fact have serendipity here with do together the change to the value form and the new meaning of "obj =".

Regarding using "<=" instead of "<", upon looking at it in the above examples, I think plain "<" looks better. Plus, "<=" when used together with "=" connotes a potentially confusing semantics. Admittedly, "<" by itself connotes a potentially misleading semantics, since if there are no specializing components, "<" does in fact mean "<=". However, on the whole, I think "<" wins over "<=", at least that's the current thinking.

So, the final bottom line is that we should do this, in conjunction with adding the `value` keyword.

Yet, Yet, ... Yet More on Types versus Values

Today's mindset: it seems pretty foolish not to syntactically distinguish between objects and values, given that they are semantically separate things. This separation, is after all, quite fundamental, since types are statically defined and elaborated whereas values are dynamically defined and evaluated. Therefore, I suggest, most humbly, that we just do it once and for all -- introduce a `value` keyword. This new keyword need not precluded partially instantiated types, necessarily, but the discussion on type spellings in

While we're at the new `value` keyword, let's just go whole hog with definition for consistency and say that the currently unnamed entities -- axioms and theorems, now get names. Hence, we'll have the following complete set of entity categories:

```
object name ...
operation name ...
value name ...
var name ...
axiom name ...
theorem name ...
```

Reminder

See below about changing the keyword pair "instance of" to the pair "inherits from". It's a good and important idea.

Caution Re. Structural Equiv and Axioms

It seems that structural equiv should *not* be used in the case of axioms that quantify over types. E.g., consider

```
obj MonthlyDate is integer;
axiom forall (md:MonthlyData) (md >= 1) and (md <= 31);
```

In this case, the hoped for meaning seems clearly to be that the axiom applies only to values of type MonthlyDate, not to all integers. Think clearly about this.

Long-Missing Discussion of Auto-Generation of Op Types

I believe that we've discussed this problem, but I do not find it explicitly explained in these notes. The problem is that the automatic generation of an obj from and op is made messy by op overloading. Viz., with two or more overloads of an op, we get overloaded objs, which cannot work exactly.

The problem with the thinking thus far in this area has been that the duality should and probably cannot be complete in this area. The statement of this would-be duality is

> *For any obj named "X", there is an (automatically-generated) op named "X" that is its constructor AND for any op named "X" there is an (automatically-generated) obj named "X" that is its op type.*

The problem, I now believe, is with the second conjoin in this statement, viz. the auto-gen of an op type *named* "X". The problem is that this two-way naming cannot really work. What's confused here is that we have three semantic concepts going on here when we need two for the duality. I.e., we have one semantic concept too many. The concepts are

a. Every obj has a dual op of the same name called its constructor, the signature of which is the obj's type.

b. Every op that has the same name as an obj and signature that matches the obj's type is the constructor for that obj.

c. Every op has a type which is its type as an op type

What we're saying here is that the last of these three concepts has to go when it comes to same naming. I.e., the op-type obj for an op should not (cannot) have the same exact name as the op. We'll talk some more about this below, but a good candidate name for this op type is the mangled name, with some kind of "Op" prefix or suffix, the prefix or suffix being necessary for opaque types where the mangled name and actual name of an op are identical.

To further clarify the matter, here are some points the complements of which are *not* true (that's twisted, eh?):

a. Not every op is the constructor for some obj.

b. Not every obj is the op type of some op.

So, what about the naming convention for auto-gen'd op types? How about

Op*<op-namee><op-inputs>*to*<op-inputs>*

E.g., for the op definition (t op Foo(X,Y,Z)->(X,Y)

The auto-gen'd op type name is "`OpFooXandYandZtoXandY`". Somewhat ugly, but (a) quite consistent with intuitive type spellings of signatures; (b) fine given how infrequently users are likely to use auto-gen'd op type names; and (c) fine because this form of auto-gen'd name is unlikely to coincidently match the name of a user-defined type, particularly if normal RSL conventions are used, which disallow the use of lowercase "and" as a name word separator.

Generalizing Relational Definitions

What have we learned from studying UML? That the RSL treatment of relations can be generalized and sanitized as follows:

a. In general, we should now sanitize things so that the syntax (and semantics) of an attribute are defined as either a value-valued expression or type-valued expression. The former can be called *value* attributes, the latter *relational attributes*.

b. Given this, we define the syntax (and semantics) of a relational attribute to be a composition expression, period. This allows both of the current cases of object name or comment, plus all of the general power of composition expressions. This admits much of the semantics of adornments that UML allows; and generally strengthens the relational semantics of RSL.

c. *All* current attributes should have a built-in attribute name, inheritance and function value in particular. The TODO list suggests "implementation" for the former, but this sucks. Also, inheritance is problematic because I can't think of a good single-word name for the attribute. "inherits from", "instance of", "subclass of" are candidates. "inherits" as a single-word name is a possibility. "instance of" should probably be ruled out given the connotation of "instance" that has taken hold. Viz., "instance" means concrete value, not subtype. Let's try *"inheritance"* and *"expression"* for single-word names. There's more discussion below. BTW, the rationale for "inheritance" is that it's a noun, which is consistent with the current naming of a built-in attributes. If we go with something like "inherits from" we've moved into verb phrases, which is not necessarily bad, but is inconsistent with current nomenclature style.

d. We should also allow the size of a list to be specified in a comp expression, using the normal syntax of list indexing, including ranges. E.g.,

```
obj X is Y[2] and Z[5..10];
```

which means that X is a tuple of 2 Y's and between 5 and 10 Z's.

e. As Porcelli noted, there now appears to be an unnecessary redundancy between the name/value notation of attributes versus the very similar-looking name/value notation within compositional exprs. This can be nicely (I think) rationalized by saying that name/value attribute pairs are meta-components, whereas name/value tuple elements are structural-components.

f. OK, so here's a pretty sanitary rule for how to determine whether an attribute is type-valued or value-valued: if there is a ':' used in the attribute definition then it's value-valued, otherwise it is not.

g. Given this, we should be able to (actually, we must be able to) define the class of all of the built-in attributes. Here's a shot at it:

   i. components: type valued

   ii. inputs and outputs: type valued

   iii. ops: (restricted) type valued (where commands are used as tuple operators, and the tuple elements are op types)

   iv. precond and postcond: op(any)->boolean

   v. where: (restricted) boolean-valued

OK, let's look at a revised nomenclature for built-in obj and op attributes:

```
object X inherits from Y and Z is A and B and C;
```

is the short form of

```
object X
  inheritance: Y and Z;
  components: A and B and C;
end X;
```

Here is a long form showing all of the built-in object attributes:

```
object X
  inheritance: Y and Z;
  components: a:A and b:B and c:C;
  operations: P and Q and R;
  equations: P(x) == Q(y);
  actions: a.a1 = z.a1;
  description: (* ... *);
  where: D = E;
end X;
```

Note here that we've dropped the "is" in the long form. Note also that a "value" attribute can be used in place of the components attribute for a fully concrete object.

Yet More on Concrete Values

Yet another possibility for solving the abstract versus concrete value dilemma is to allow only constant values in object value expressions, where constant values do not include (some) expression operators. This makes sense in general, given strong typing. Here are some more thoughts, rapidly.

We've always liked the ideas of partially concrete types, where, e.g., components could be something like "integer and 2 and integer". I.e., one or more components can be concrete values. If we go with only allowing constant values syntactically as values, we can probably go back to the syntax that allows composition expressions to be mixed types and values. Recall that then problem with this syntax is that operators like "and" are ambiguous in a composition expression that allows both value and type expression operators.

The notion of a *fully* concrete object is that all of its components are values, not types. This is consistent with the nomenclature used in the type spelling rules where we say that the spelling of a value is "the ...".

BOTTOM LINE for this discussion: It appears that the current solution of allowing value *identifiers* in type expressions is probably the best one. However reasonable it may seem to disallow boolean operators in constant expressions, this really is an arbitrary restriction. After all, we cannot disallow *all* expression operators in value expressions, since this would prevent constructing concrete composite values with list and tuple operators. Given that we must admit *some* expression operators in concrete value defs, it can only be arbitrary what ones to eliminate after that. When we think about it, it really is just a syntactic coincidence that "and" has two meanings in type versus value expressions.

So I think things are just fine if we stick with the current syntax that allows value identifiers in type expressions, but does not otherwise allow mixing and matching of type and value expressions.

I believe it's also OK to continue with the short form "is" versus "=" notations. The former is for any form of abstract object, including partially concrete ones. The latter notation is reserved only for fully concrete objects. Syntactically, this is what allows us to segregate type and value expressions.

As hinted above, we might now consider dropping the "is" from the long form of definition. This will leave "is" strictly as short form syntax. I think I like this.

Given the graceful transition from abstract to concrete values, the components attribute should be usable for both the type and value parts of an object. Here's the ground rule for this: *An object identifier denotes either a type or a value, never both.*[1] Hence, the components attribute can be used to hold an object's type or value, since an object will never have both at the same time. There is a potentially annoying syntactic problem with this, which has actually been around almost forever. Viz., the plural noun "components" seems an inappropriate name to used for the single value of an object, just as it does for a single type. E.g., if we define type X as identically type integer, the usage "components: integer": has always been strained. Similarly, if we define X as identically the integer 10, the use of "components: 10" again seems strained.

To some extent, at least formally anyway, we've alleviated the "components" as singular attribute problem by defining a one-tuple type to be the same as component type itself. This solves the problem for values as well. However, for something as fundamental as this, it seems hokey to have such a severe nomenclature problem.

Since we have worked out the semantics satisfactorily, we should be able to work out some accompanying notation. How about this: (1) we'll allow both the singular "component" as well as the plural "components" as a nod to objects that have exactly one component; we could even have the checker check for this; (2) we'll allow "value" to be substituted for "component(s)" when an object is a fully concrete value; the checker should in fact check for this, since we'll be evaluating the attribute differently in the two cases (i.e., type versus value expression). I was just about to say that I'm not even sure about the "value" attribute, since "component: 10;" would be as good as "value: 10;". However, we must recall the ambiguity between type and value exprs, hence the necessity for "value" as a separate attribute name. After all of this is done, if the user still has some doubt about the nomenclatural clarity of specifying type identity with a "component", we can sugar up the "one tuple of X = X" argument and take care of things pretty well.

We have toyed with the idea of a *default* value for an object, which could be present in addition to components. I'd rather not build such a thing in. Rather, we'll just specify an `Initialize` operation that creates objects and specifies certain concrete values objects must have for starters. See the next item for some further interesting discussion about this.

FLASH on Specifying Read-Only Objects

To make an object read-only, with a given concrete value throughout execution, specify it with an axiom as follows:

```
obj X is ...;
obj DefaultXVal = ...;
axiom forall (x:X) (x = DefaultXVal);
```

Hmm, this is pretty cute.

---

[1] As noted on 24nov97 in ../ref-man/formal-type-rules.me, this ground rule needs to be reconciled with the current write up of the formal type rules.

(Final) Decisions for Version 4

In light of preceding discussion, this item is bogus.

Oh, please, say this is really the last word on the subject. We now *will* goto a separate value keyword, instead of the 'is' versus '=' object notation. Here's how we'll explain it.

There are two main kinds of entities -- *objects* and *operations*. Objects denote data types and operations denote functions. In addition, there are the following ancillary kinds of entities:

- *values*, which denote constant object values of some type
- *variables*, which denote global state variables of some type
- *axioms*, which denote definitional predicates that are globally true
- *theorems*, which denote postulated predicates that should be proved true with respect to axioms
- *attribute definitions*, which are meta-definitions used to define new entity attributes

Here is some further discussion/explanation to rationalize the keyword syntax, in particular. In V4, let's have 'is' and '=' be synonyms. This sounds pretty good, even though it isn't quite in keeping with the so-called *is-a* versus *has-a* distinction. We can easily, I think, rationalize this with the "hey, we're in the specification language, not programming language business" argument. Given this, in V4 here's a revised bit of syntax table:

keyword	**equivalent symbol**	meaning
is	=	is defined as; aka, is composed of
instance of	<	inherits from

This seems pretty clean syntactically, since it allows defs of the form

obj X < Y = A and B and C;
obj Y instance of Z is D and E and F;

Further rationalization goes like this. The reason we use the keyword "value" instead of "constant" is again because we're a specification language, not a programming language. This seems to be shades of ML, which is fine. (Global) variables break the functional semantics, but are considered necessary in order to specify certain state-based semantics, including concurrency based on shared variables.

Well, We Gotta Keep Doin' Better

In thinking about how to resolve op inheritance, it occurs that we are not doing obj inheritance properly. In particular, in order to be able to do struct equiv properly, we need to do the following:

- Define the order in which parent components are snatched.
- Allow duplicates, except of the same name.

Things to See in Test Files

See:

- rsl/alpha/newtests/one-tuple-niceties.rsl
- rsl/alpha/newtests/generic-sort-function.rsl

Yet More on "obj X = ..." Versus "value X = ..."

Farg -- all of the crap below is yet another circle. Try this. A definition of the following form

```
obj X = "abc";
```

defines *both* a value and a type. Contrary to what we may have been thinking, however, the type of X is *NOT* string. Rather, its type is X, which is a unique opaque type with is a subtype of string. It seems that we may have had this idea before, but were unable to make it work (except maybe in the "THE, Ultimate" discussion below). I think the reason is we didn't have one-tuple-of-X = X type equiv rule in place at the time.

To unify things completely, it might be nice to go back to the old, old idea that values can be first-class operands in components exprs. However, this leads to very serious overloading problems, as in:

```
obj x is a and b;
```

wherein we cannot tell if obj x should be a tuple type or a boolean type. I.e., one interpretation of "a and b" takes "and" to be tuple construction, whereas the expression interpretation of "a and b" takes "and" to be boolean conjunction. This wont do. Therefore, I think we need to live with the slightly hokey

f(s:Sex)->boolean = s = "male";

OK, so we go with "value =" instead of "object =" -- what really do we buy? Seems to me that if we don't really solve any big problem, might want to stick with "object =" simply for backward compat and notational orthogonality. So, the question remains, do we solve any big problem simply by subbing "value" for "object"? As I speak, I'm getting the feeling that the answer is no. Essentially, what we're coming to is that an "obj =" definition does not in fact define a type at all, but rather a (const) variable. Hence, the stuff about "restricted" type need not exist. So what we're left with is simply a keyword thing. Viz., should be use "obj =" or "value ="? Is this really all there is to it? Let's play with the implementation and see, since we now need to fix the problem with rsl3/tests/dan-lawyers-and-judges.rsl.

OK, now we've implemented a bit. To be clear, what we've just implemented is that the "obj X = ..." form now defines X as of class C_Var, not C_Obj. This makes things pretty darn clear cut semantically, but has lead to some new issues. One consequence of leaving the "obj =" form instead of the semantically potentially more descriptive "val =" form are the error messages that say "Components of objects must be objects (X is not)". This error message will happen if X is a concrete value, which is quite confusing to the hapless user. Hence, for now, we've dealt with the problem by changing the text of this error message "Components of objects must be *abstract* objects (X is not)". A little hokey, but maybe we can live with it.

Another potentially more troublesome consequence of the new C_Var implementation is what's happened to the tests/bnf.rsl example. Recall that this example is cited below in the nicely concluded discussion of symbolic literals. What's happened now is that bnf.rsl has gone bad (again?). Specifically,

```
obj Operator is PlusSign | MinusSign | TimesSign | DivideSign;

obj PlusSign = '+';
obj MinusSign = '-';
obj TimesSign = '*';
obj DivideSign = '/';
```

now produces four error messages since PlusSign, etc. are now concrete objs (a.k.a. vars), which are not legal in comp exprs. At the same time, bnf-v2.rsl is (still) OK. It contains just

```
obj Operator is '+' | '-' | '*' | '/';
```

The seemingly ugly part here is that "equals-for-equals" semantics appears to have broken down, at least superficially. Viz., '+' is OK in directly in a comp expression, but not indirectly via having been defined as the value of a concrete object. I believe that the semantics are still sound here, in that the symbolic literal notation is define clearly as a special form of designation that is the *only* notation that designates *either* a type or value, depending on context. Given this, the declaration "obj PlusSign = declaration, PlusSign has now become a concrete value only, and since idents do not have the special dual designation property of sym lits, the use of such a concrete object ident such as PlusSign will (and should) be illegal in a comp expression. All of this reasoning not withstanding, I'm starting to think at this point, after this latest round of implementation, the that "value =" notation might now be better, since it clearly denotes when something is a value and not, therefore, an abstract object. Hmm., need to cogitate some more.

Decisions, Decisions

Once and for all, let's do the following:

1. Throw out the syntactic form "obj X = ...", replacing it with "value X = ...". (Well, as can seen above, we're still thinking about this one.)
2. Implement the short-hand usage if 'E' lists in comp exprs.
3. Implement full struct type equiv and the equivalence of obj exprs and ins and outs. (As a practical matter, we may never get to this; rather we'll just implement as much of it as necessary to get auto decl of constructor ops, and whatever other nice effects we need. We're still thinking about it as of now.)
4. Implement the "top" opaque type any, from which all types implicitly inherit. This will obviate the need to implement '*' as a type, I think. The only op defined on any will be '=', which is always false for function types. (Still thinking about this one too. Some recent work with opaque types as vars (in particular in newtests/master-list.rsl) may obviate the need for a true top type.)

Concrete Values, Rere...revisited

It seems to me that about the only reason we've been clinging to the "obj is" versus "obj =" notation, instead of an explicit value keyword, is for enum literals as strings. It seems further that the idea of enum literals as strings has pretty much gone away. Therefore, I don't think we need to mess with the idea of a "restricted" type as we were below, since I recall that this stemmed from the whole enums as strings mess.

Therefore, if we do stick with "obj = ..." instead of "value = ..." it's for notational orthogonality, not because we want to stick with restricted types or strings as enum literals. See discussion about the latter to follow.

The (Long-Standing) Trouble with Enum Lits

In PLs, there is are generally two separate constructs for enums versus unions. In RSL, there is the single **or** for both. This, it seems, has led to us going around in circles for so long about how exactly to represent enum lits. Let's see if we have a clarifying summary of where we've come to at this point, which is hopefully a sound solution to the problem of how to represent enum lits.

First off, enum lits are NOT strings and NOT integers. Actually, enum lits have two identities -- as types and as values. As a type, an enum lit is an opaque type. As a value, an enum lit is a constructed value of an opaque type, which is fundamentally denoted as E(), for opaque type E. Since enum lits are an oft-used construct, the syntactic sugaring of 'E' is provided.

Conceptually, the following rule may seem in order:

> An opaque type E is a type, NOT a value; an enum lit 'E' is a value, NOT a type.

This leads to the more general rule that

> No single denotation can represent both a type and a value.

A, perhaps the only remaining, problem with this is exemplified by the bnf-v2.rsl file, where enum lit values are used as type names. The problem with this bit of notational trickery is that it violates the preceding general rule, and leads back to the idea of restricted types, which we've never fully formalized properly. (But see bottom line below.)

A conclusion could be that we could probably live with the preceding rule, if it gives us a simple and sound semantics, even if we don't get quite as nice a bnf notation. Since BNFs in RSL are presumably not that common, this probably isn't a major problem.

However, if we want to allow the convenience of the notation in examples like bnf-v2.rsl, a (the?) fundamental question is:

> Can we devise a sound semantics that will support enum lit designations being used as both types and values?

Here's a crack at how we might implement the type checking of such:

a. Whenever a 'E' lit appears in a type expression, enter an opaque type of that name ("E") in the Level 0 symtab if one does not already exist.

b. Complain clearly if the enum lit is already def'd as a non-opaque type.

c. Whenever 'E' appears in a value expression, it denotes a value of type E.

d. Be sure to clarify in the users manual that 'E' is never = "E".

Actually, this may be a pretty sweet little solution to the enum lit thorn in the side. Viz. the following works fine now:

```
obj Sex is 'Male' or 'Female';
```

and doesn't even look half bad. Also, it looks like it can be explained pretty easily. Hmm., maybe we've fixed things finally.

Also, bottom lineishly, except for idents 'E' lits, no denotation is both b type and a value.

Why 0-ary Constructors for Non-Opaque Types are NOT Senseless

WRONG REASONING: Because such a constructor would always have to create a nil value, but given that nil is totally overloaded anyway, one might as well just use nil in the first place.

CORRECT REASONING: Because such a constructor will NOT always create a nil value, but rather for tuples will create a tuple value with potentially initialized fields, per the object definition. See note below about nice duality between obj's and op's vis a vis init'd name/type pairs.

On Generic Instantiation

See the very important discussion in newtests/one-tuple-niceties.rsl.

On Subtyping Atomic Types

If we've not already, we need to decide if this is allowed. Apparently the 3.1 type checker thinks it's OK, since it passes tests/subtype-poly-tst.rsl, q.v.

On One-Tuples

As noted under the "POSSIBLE FLASH" heading below, we can hopefully implement the following rule: there is no distinction between a one-tuple of type T and type T itself. We'll talk more about this, and its lovely consequences, soon.

On Generic Functions

Consider

```
function IsSorted(l:*)
    forall (i,j: integer |
        (i in [1..#l]) and (j in [1..#l]) and (i<j))
            l[i].id < l[j].id;
```

The idea is that allowing '*' by itself to be a type designator, we're writing a generic function that can be applied to any list. Now, what happens if we overload this? Well, I think the same basic "lowest" rule for overload resolution can apply is with subtype polymorphism. Viz., such fully generic list functions are the "highest" in the chain, and will be overridden by any other functions of the same name that take more specific forms of lists.

We might even consider that any list object is automatically a subclass of generic type list, which means that it inherits things that are true for all lists. We could make something conceptually nice and uniform out of this, if we could get the different forms of instantiation done straight.

On the practical side, here's a sketch of the (seemingly ML-like) way that such generic (list) functions would have to be type checked:

1. When we type check the generic itself, the base type of the list will have to be some type var, which checks like type Any (i.e., compat with everything, except maybe function types, but we'll have to see about this).
2. At any call to the function, we'll have to (re)instantiate the function's body, with the appropriate generic instantiation.
3. We need to give much better error messages that ML does, obviously. Something like the following would be nice:

```
Generic function invocation fails because of the following type problems:
    ...
```

   after which we simply proceed to list the error messages that normal type checking on the instantiated body produces, probably with line numbers for the cite of the generic definition.

Conceptual Drift

NEW DISCUSSION:

The argument under "OLD DISCUSSION" below is evidently wrong, or at least confused. Contrary to what's said there, it seems that it *is* OK to use opaque types as type vars, particularly if we use the rule that {T} and T are the same type. Here's the deal.

Where the discussion below goes wrong is in the following statement:

> The earlier position was to treat opaque types as type vars, *such that any type could bind to an opaque type*, modulo varname use in a signature.

In particular, it's the italicized part that's wrong. Viz., we do not let any type bind to an opaque type, only explicitly declared subtypes can so bind. And this is precisely the normal subtype polymorphism rule -- that any type T1 can bind to any type T2, iff T1 <= T2.

Hence, we're OK with considering opaque types to be a *form of* type var, but not the fully general form of type var, as in ML.

OLD DISCUSSION:

It seems from the discussions below that we've switched positions diametrically with respect to the equivalence rule for opaque types. Viz.,

- The current position is that each distinct opaque object represents a distinct type, not equivalent to any other type. This is in fact what the type checker enforces. The earlier position was to treat opaque types as type vars, such that any type could bind to an opaque type, modulo varname use in a signature.

Type latticewise, the current position has opaque types on the bottom of the lattice, whereas the earlier position has them at the top.

What this means at present is that we should stick with the current position, but add a syntactic notation for type vars, if necessary. We discuss the type var issue below.

Type Vars Revisited

So, the question is, do we need type vars, and if so for what? While we're at it, we need to fix all of the generic instantiation crap that's in the ref man. In particular, when thinking about type vars in generics we probably don't want to have where clause substitution types to have to be subtypes of what they substitute for (see the ref man).

The point of confusion is to exactly what extent the seemingly true statement "inheritance" is not "generics" is true, particularly in RSL. Let's look that the following very familiar example, the current where-clause way:

```
obj Stack is Elem*
    ops: Push, Pop, Top;
end Stack;
```

```
obj IntStack < Stack
    where: Elem = integer;
end IntStack;
```

Let's try a more conventional parameterized type way:

```
obj Stack(Elem?) is Elem?*
    ops: Push, Pop, Top;
end Stack;

obj IntStack is Stack(integer);
```

A key problem we're suffering with is the nature of inheritance as tuple creation. Even if a subobject does not add new components, it *automatically* becomes a tuple. I think this needs to be reconsidered.

A promising-looking idea would be to eliminate the immediate conversion of a subtype to a tuple. Rather, wait until the second inherited or defined component is found before converting to a tuple.

POSSIBLE FLASH (see full discussion above): make no semantic distinction between a one-tuple and a non-tuple. E.g., an object of type integer is the same as an object of type {integer}. One nice benefit of this approach is that the existing anomaly of not being able to access a single-tuple elem via '.' its name would go away. The rule would be that a one-tuple could be accessed with or without a field name selector. Hmm, this seems pretty-darn nice so far.

Another mess we've gotten ourselves into is not knowing the difference between generic instantiation via '<' versus 'is'. Consider the following example (excerpted from tests/stack.rsl):

```
obj Stack(<<Elem>>) is <<Elem>>*;
obj IntStack is Stack(integer);          (* BIG QUESTION: what's the diff *)
obj InsStack < Stack(integer);           (*  between these two?? *)
```

Lest we think where-clause notation might fix things here, it doesn't. I.e., we can have a where clause with either '<' or 'is'.

Here are some pertinent questions related to this issue:

1. Should both forms of instantiation be allowed?
2. If we use the parm'd types notation, should an instantiation form be allowed anywhere that a type is allowed, or should we allow it only in a parts_spec? FORGET THIS: this wont make any diff if op sigs are now the the same as parts_specs.

More on Op Args

In an effort to define automatic constructor ops, we need yet again to address precisely how to declare op args. We have discussed at length that we would like to allow a fully general parts_spec in the input and output fields of an op. To do this, we need to clarify if we have the ML-like single-arg model, or if not, exactly what we do have. Let's try the following:

- A full-on parts_spec can in fact be given for op ins and outs.
- Each *top-level* **and** operand is treated as a separate arg.
- If the parts_spec is a single list, then the function is n-ary.
- If the parts_spec is an **or**'d expr, then the function has exactly one arg of the **or**'d type.
- Syntactically, we'll fully unify parts_spec and ins(outs)_parts_spec. Probably the easiest way to do this is to use parts_spec everywhere, with the addition of using init_name_type_pair in place of name_type_opt_pair. This has a seemingly nice duality effect, in that we can interpret an initializer in a obj expr to mean that whenever a concrete value of the type is created, the initialized field(s) will have the given value(s). This does seem to have a serendipitous duality effect, in that we are blending objects and ops quite nicely now.

Casting by Constructor

The following discussion is historical only in that we have decided to be happy with '<' as the instance-of operation. We'll also add subclass (of) for some additional syntactic sugaring.

We'd like to use ':' as the instance-of operator. This leads to an syntactic ambiguity with uses of ':' to declare type in a concrete obj decl. This, somewhat serendipitously, suggests that we should eliminate the latter use of concrete object of a particular named type. This leads to the idea of an automatically defined overload of object constructor that takes an argument of the structured type of its components. Ah, but we probably already have this.

A motivating example for this is the biggie burger food menu, which in the past we have declared something like this:

```
obj DefaultMenu:FoodMenu = [ ["Burger", 1.89, [...]], ... ];
```

If we eliminate this use of ':', we get the following:

```
obj DefaultMenu =
    FoodMenu( MenuSection("Burger", 1.89, Acessories(...)), ... );
```

which probably looks better mnemonically anyway.

There seems to be really little problem with eliminating this use of ':', since it's only of consequence in concrete obj decls. In other binding contexts, we'll have a declared formal (LHS) to determine the type, and in effect do the casting for us. The "casting" to which we refer here is that of casting a structured value specified with the generic bracket constructors into a specific named type.

All of this is related, yet further, to the next heading on concrete values.

Concrete Values, for the Umpteenth Time

I thinking again that we should add the keyword "value" to the language. Ah shucks, I wish I could make up my mind about this! Maybe we can make "let" be a top-level (i.e., module-level) construct, so that values can be created that way. E.g., now we have:

```
module Foo;
    object X = 10;
end Foo;
```

With a value keyword, we'd have:

```
module Foo;
    val X = 10;
end Foo;
```

Allowing top-level let, we'd have module Foo;
let X = 10; end Foo;

I kinda like that last one (the let). Let's think about it (pun partially intended).

New, Improved Import/Export Rules

Summary:

- Legal syntactic forms:
  o import M, for module named M
  o import X, for export X
  o import M.X, for module M, export X
  o export X, for defined symbol X
  o export all, for "all" a keyword
- old forms now axed:
  o from M import X
- import M, for M a module name, imports all exports from module M
- import x, for some export x, imports x from the first (only) module that exports it; see below for error cases (i.e, if 0

or >1 modules export x)

- export all has the obvious meaning
- import M.x should be a legal form, with obvious meaning
- the qualification/redundancy/insufficiency business should be handled as follows:
  - *o* All imports should be unqualified by default whenever possible; i.e., reference to any import should automatically be allowed in unqualified form, as long as the imported ident is not already defined in the importing scope.
  - *o* If an imported ident is the same as an existing ident, then the checker issues a warning indicating that qualification will be required for that ident.
  - *o* If > 1 module defines an import, then the checker issues a warning listing each of the modules that exports the ident, and indicates that each version will be available in qualified form only.
  - *o* Note that a warning or error is not issued at the time of export for a module that exports an ident for the second time or beyond. Rather, the warning is used at the time of import. This is because multiple export is not really an error at all, since it can be dealt with by explicit qualification, either at each use or at import.
- If 0 modules define an import, then the checker issues an error to that effect.

RSL-like syntax for DEMO interface modules

Given below is an adaptation of the following DISL code:

```
INTERFACE DummyName{
    Obj1 := DrawRect(342, 110, 472, 142);
    Obj0 := DrawRect(88, 112, 218, 144);

    MAPPINGS Obj0:
       LEFTDOWN: op0;

    OPERATION op0(s : Selection; l : integer; b : integer) {
       STIMULUS MoveSelection(s, l, b);
       RESPONSE {
           MoveSelection(Obj1, integer(3.250000 * real(l)),
              integer(1.600000 * real(b)));
       }
    }
}
```

Here’s the RSL-like equivalent (see ../../rsl/alpha/newtests/sample-demo-code.rsl):

```
(* interface *) module DummyName;

    from Demo import Selection;

    define object attribute leftdown;
    define operation attribute stimulus, response;

    object Obj1 = DrawRect(342, 110, 472, 142)
       leftdown: op0;
    end Obj1;

    object Obj0 = DrawRect(88, 112, 218, 144);

    operation op0(s:Selection, l:integer, b:integer) < DemoOp
       stimulus:
           MoveSelection(s, l, b);
       response:
           MoveSelection(Obj1, integer(3.250000 * real(l)),
              integer(1.600000 * real(b)));
    end op0;
```

```
end DummyName;
```

where module Demo is:

```
module Demo;
    export Selection;

    object DemoWorld;
    object Selection;
    operation DemoOp(DemoWorld)->DemoWorld;
end Demo;
```

Note that this requires that we expand RSL syntax to allow attribute values to be exprs, which we've done. This, BTW, lead to the new reduce/reduce error, which we've determined to be OK.

Yet Yet More on Func Call Type Checking

The latest idea is that we may be able to have MLish parm bundling and unbundling, as long as we do it at the top-level of parm nesting only. Consider the following familiar-looking defs:
.(t
obj X is A, B, C; op F1(X)->(X); op F1(X)->(A, B, C) op F2(A, B, C)->(X) op F3(A, B, C)->(A, B, C);

NEED TO EXPAND THIS, looking carefully at what's already been said below.

There are also some important, seemingly deeper thots on subtype poly vis a vis constructed types, e.g., with "[" ... "]" and "{" ... "}". If it's not completely clear in what's said below, these constructors construct types that are structurally compat with list and tuple types, but *NOT* subtype compat with *ANY* types. This is because these constructors build anonymous types, which cannot, by definition, be subtype compat. It seems like what could be emerging here is a version of name type equiv that relates to subtyping. I.e., we have a structural equiv rule at the level of straight types, but structural equiv does not extend into subtypes. Probably should get off butt and see what research there is about this out there. It seems likely that this kind of rule already exists, maybe from some time ago.

Type Any, One More Time

Thot: provide pre-def'd type vars "any", and "anyn", for all n, with the intent that any1, e.g., used in a signature is (must be) the same type in all signature occurrences, but is not necessarily the same as, e.g,, type any2. Drawback here is using up ident names, but this may not be a problem. Need to think about it. Alternative, used in scattered examples, is to use '?' an ident prefix or suffix to denote a type var. Prob with this is lexical/syntactic confusion with '?' as the infix tag selection operator.

On the Ultimate Generalization of Type Checking

It may just be possible to have it all, viz., to have full subtype polymorphism; structural equivalence; complete compatibility between comp exprs and op sigs; and overloading. Here are some observations and rules in this regard, which need further discussion,

1. A la ML, we can in theory consider all funcs to be of a single arg.
2. Consider, therefore, using (,...,) the generic tuple constructor; [,...,] might still be the generic list constructor, but we might even want to go for (,...,) for list construction as well. This, it seems, will help unify the concept of comp exprs and signatures being the same thing. Note that we might also want to leave { ... } and [...] in, and just have some less-than-perfect rules for how they can be omitted in the context of an op call. E.g.,
   ```
   op Op(integer and string)->integer;
   op main() = (
       Op(1, "abc");    (* OK *)
   ```

```
Op({1, "abc"}); (* Also OK, and equiv to the preceding. *)
```

Note that things get funky in cases such as the following, wherein we'll need some clear disambiguating rules:

```
obj ThreeInts is integer and integer and integer;
op Op(integer and integer and integer and ThreeInts integer*);
op main() = (
    Op(1,2,3, 4,5,6, 7,8,9,10)
);
```

3. To avoid absurdly difficult, if not impossible inference, an object constructed with the generic tuple or list constructor operators, *CAN NEVER BE* considered to be of a subtype, since there's no name associated with it. Hence, to enable subtype inferencing in parm binding, a named constructor op must be used. E.g.,

```
obj X is string;
obj Y < X is integer and integer;
op DoX(X)->X;
var y:Y;
axiom ... and DoX("abc",1,2) and ... ;
           (* ^^^^^^^^ This wont work since ("abc",1,2) is in fact compat with
            * type Y, however it cannot easily be inferred to be a Y in this
            * context, and hence a subtype poly match of DoX wont happen. *)

axiom ... and DoX(Y("abc",1,2)) and ... ;
           (* ^^^^^^^^^^^^^^^^^ This *will* work since Y("abc",1,2) is directly
            * discernible as a Y, and hence the subtype poly match can be
            * found. *)
```

Single-Tuple's Revisited

Despite observations to the contrary, we may well want to allow single-elem tuples to be equiv (say, structurally) to types of the elem type. The reason is a good one, when one considers the existing examples of generic list-structured objects, such as the ref man generic DB example. Consider:

```
obj GenericList is Elem*;
obj IntList < GenericList
    where: Elem = integer;
end IntList;

op f(l:IntList, i:integer)->boolean =
    l[1] = i; (* !! Wont work, because IntList is of type (integer* and)
              * NOT of type integer*.  Oops -- this ain't so nice. *)
```

The reason, I believe, that we were clinging to non-equiv of single-elem tuples and the elem type is to cling to name type equiv, which in turn was, among other possible reasons, to allow op ordering dependencies to be specified by unique types. We probably want to avoid this anyway, but if it's really necessary, say in a comm protocol specification, we can do it via unique opaque types, since these are always unique. I think we'll be hosed pretty badly if we say that all opaque types are equiv, so we can rely on their uniqueness. While this may make the spec of sequential control dependency a bit uglier than it might otherwise be, we can live with this since we want to frown on control dependencies anyway.

More on Op Inheritance

The stuff below about reintroducing direc op inheritance still seems good. We need to make the following refinement. An op definition will define a type all right, but a value-constrained type. The subtle problem we need to deal with is an invocation of subop, bound to a var of a parent type. E.g., in the example below we have:

```
op class GenericGraphicsOp(Canvas)->Canvas;
op Move(SelectClassPaZOrms) < GenericGraphicsOp;
```

```
    op Scale(ScaleClassParms) < GenericGraphicsOp;

op ExecuteSelectClass(gop:GenericGraphicsOp, parms:GenericGraphicsOpParms) =
    gop.<Move(parms.<SelectClassParms);

op main() = (
    let s = Selection(...);
    ExecuteSelectClass(Move, SelectClassParms(s, 10, 20));
```

The call `gop.<Move(parms.<SelectClassParms)` is just fine in this context, since `gop` is bound to op `Move`. Suppose, however, that the body of `ExecuteSelectClassOp` was

```
gop(parms);
```

The deal here is that we can't know that `gop` is bound to `Move`, yet at runtime we must be able to run something. So, we'll have some *behavior* inheritance. What this means is than subops inherit the *body* of their parent op. It seems that the runtime rule can be that any subop will inherit the body(ies) of all parent ops, an analogous manner to how subobjs inherit fields. Then, in situations such as the above, when a subop instance is hanging out in a parent-type variable, the inherited parent op body can be executed by extracting it from the subop. Cool.

In general, the concept of behavior inheritance should *always* be on, to be precisely the dual of structural inheritance. To be precise, behavior inheritance will work akin to constructor behavior inheritance in C++. !BUT WOA HERE -- this can't work in a functional setting, since we seem to have no way to capture the return value of parent execution. So, maybe what has to go on here is to execute only parent behavior in cases such as the above where we have a subop value hanging out in a parent var. !NO BUT WOA AGAIN -- maybe we do in fact have a way to get at the return value of the parent. Viz., it's in the output parm(s) of the parent. E.g., adding parm names to the op defs above:

```
op class GenericGraphicsOp(c:Canvas)->(c':Canvas);
op Move(scp:SelectClassParms) < GenericGraphicsOp;
op Scale(scp:ScaleClassParms) < GenericGraphicsOp;
```

Given this, we can refer to the output of GenericGraphicsOp as a parent op from within the definition of one of its children, say Move, as in:

```
op Move(scp:SelectClassParms) < GenericGraphicsOp =
    c';
```

which simply means that Move returns just the c' that its parent GenericGraphicsOp returned. (This could be *way* cool, if it pans out.) Move could do more if it chose, including not referring to c' at all, in which case c''s return value from Move would be whatever its parent computed it to be. If Move wants to do more with c', then it can, as in

```
op Move(scp:SelectClassParms) < GenericGraphicsOp =
   f(c');
```

for some function f.

THE, Ultimate, For Sure, Absolutely Final Ruling on Union Types (Maybe)

OK, let's see if we can define some type checking rules that will make sense out of all of the possible ways that unions can be used, probably most particularly as enumeration types.

Rule 1: If all of the components of a union type are unique types, then the checker can infer the necessary injection in binding contexts.

Rule 2: If all of the components of a union type are the same type, then the checker can infer the necessary injection in binding contexts, which type is simply the component type. This rule will make the old-style enumerations work as expected.

Rule 3: If component types of a union are neither unique nor the same, then no injection inference is possible. *PERHAPS* we want to disallow this case altogether.

To make all of this work, how bout if we define concrete obj defs to mean the following.

```
obj x = val
```

<==>

```
obj x < typeof(val);
obj 'val' < typeof(val);
```

where val must be a const expression. E.g.,

```
obj x = "xyz"
```

<==>

```
obj x < string;
obj 'xyz' < string;
```

Do think about this some more.

Hmm, Really Great Looking Polymorphism May Not Be that Easy

OK, from the real demo.rsl example, here's what we'd like to do:

```
(*
 * Primitive Graphic Operations
 *)
op Select(Canvas, Selection, Location)->Canvas;
op Move(Canvas, Selection, Location)->c':Canvas;
op Scale(Canvas, Selection, ScaleFactor)->Canvas;
(* Etc., ... *)

obj Location is Coord and Coord;
obj Coord is integer;

(*
 * Generic Graphic Op Parm Classes
 *)
obj class GenericGraphicOpParms is Canvas;
obj SelectClassParms < GenericGraphicOpParms is Selection and Location;
obj ScaleClassParms  < GenericGraphicOpParms is Selection ScaleFactor;

(*
 * The Generic Type of All Graphic Ops
 *)
obj GraphicsOperation is op(GenericGraphicOpParms)->(Canvas);
```

Given such defs, can we reasonably expect the following form of application to work?

```
op ExecuteSelectClass(gop:GraphicsOperation, c:Canvas,
      s:Selection, x:Coord, y:Coord) =
    gop(c, s, x, y);

op main() = (
    let s = Selection(...);
    ExecuteSelectClass(Move, s, 10, 20);
);
```

The key questions/points here are:

i. Is the type of operation Move, as it is defined, compat with type GraphicsOperation? If so, can we define a sound bundling/unbundling scheme that will let us do what we hope for?

ii. If so, what kind of bundling/unbundling deal have we done, and how might it adversely affect overloading, in particular by increasing the amount of work we have to do to check for equivalent bundled versus unbundled proc defs during op definition checking?

Let's look a little closer at how things might work out here:

type of GraphicsOperation =	op(GenericGraphicsParms)
	>op(Canvas, Selection, Coord, Coord)
	=op(Canvas, Selection, Location)
type of Move =	op(Canvas, Selection, Location)

The hard part here seems to be the ">" inference step. Another way to look at things is that from a vanilla definition like

```
op Move(Canvas, Selection, Location)->c':Canvas;
```

we can't necessarily infer

```
op Move(GenericGraphicParms)->c':Canvas;
```

While we probably don't need to infer this at op definition time, we do at binding, which may still be troublesome, if not impossible.

Another issue is that whatever resolution algorithm is used to check bindings will also need to be used to check redundant overloadings.

We might try to ameliorate the situation be defining

```
op Select(SelectClassParms)->Canvas;
```

as an overload of Select, i.e., leaving the original definition of Select in place. However, in order to have the kind of bundling we need to make the other stuff work, this should probably be a redundant over load of the extant, i.e., of

```
op Select(Canvas, Selection, Location)->Canvas;
```

Hmm, things dont look real promising at this juncture.

We could brute force it as follows:

```
op ExecuteSelectClass(gop:GraphicsOperation, c:Canvas,
                                  s:Selection, x:Coord, y:Coord) =
    gop.<SelectClassOp(c, s, x, y);
```

but we don't have a SelectClassOp. Also, even if we did, is this too gross and is it type safe?

OK, let's just try it in a more doable way and see if even that works:

```
(*
 * Primitive Graphic Operations
 *)
op Select(SelectClassParms)->Canvas;
op Move(SelectClassParms)->Canvas;
op Scale(ScaleClassParms)->Canvas;
(* Etc., ... *)

obj Location is Coord and Coord;
obj Coord is integer;

(*
 * Generic Graphic Op Parm Classes
 *)
obj class GenericGraphicsOpParms is Canvas;
obj SelectClassParms < GenericGraphicsOpParms is Selection and Location;
obj ScaleClassParms  < GenericGraphicsOpParms is Selection and ScaleFactor;

(*
 * The Generic Types of Graphic Ops
```

```
     *)
    obj GenericGraphicsOp is op(GenericGraphicOpParms)->(Canvas);
    obj SelectClassOp < GenericGraphicsOp;
    obj ScaleClassOp < GenericGraphicsOp;


op ExecuteSelectClass(gop:GenericGraphicsOp, parms:GenericGraphicsOpParms) =
    gop.<SelectClassOp(parms.<SelectClassParms);

op main() = (
    let s = Selection(...);
    ExecuteSelectClass(Move, SelectClassParms(s, 10, 20));
);
```

Hmm, we still seem to have a fundamental problem here. The cruxt of the problem is that given the following defs

```
obj O1;
obj O2 < O1;
obj OpType is op(O1);

op F(O2);
```

it seems that it cannot be the case that F (as a value) is compat with type OpType, even though the signature of F is (subtype) compat with the signature of OpType. This lack of compatibility seems to pretty well nix the kind of generic op creation we're attempting above. For some concrete data in this area, see the C++ attempt at doing this, in ~/c++/demorsl.c. It appears that this is a genuine problem area, since we get the following extremely interesting at the point where we try to bind Move (the function value) to a parm of type GenericGraphicsOp:

```
warning: contravariance violation for method types ignored
```

Fascinating.

Here's a flash. Perhaps we want to go back to the idea of operation subclassing, with an interpretation that args are anded into subclasses. This seems like it might be the key to the idea of an op type that allows more args in its subtypes. E.g., try this:

```
    op class GenericGraphicsOp(Canvas)->Canvas;
    op Move(SelectClassParms) < GenericGraphicsOp;
    op Scale(ScaleClassParms) < GenericGraphicsOp;

op ExecuteSelectClass(gop:GenericGraphicsOp, parms:GenericGraphicsOpParms) =
    gop.<Move(parms.<SelectClassParms);

op main() = (
    let s = Selection(...);
    ExecuteSelectClass(Move, SelectClassParms(s, 10, 20));
```

Hey, I think this, at last, has some promise. What we may well be opening the door to is that an op def automatically defines a type, in a dual way to an obj automatically defining a (constructor) op. E.g., check this out:

```
op X(integer, integer)->integer;
```

automatically defines the type

```
obj X is op(integer, integer)->integer;
```

Woe, this is getting a little scary, but maybe really cool. Given the remarks above about ops defining values, evidently, it appears that an op definition defines all *three* kinds of entities -- type, value, and op. Hopefully we can use context to sort out the name overloading. Hmm, this is getting quite interesting indeed. [But as we've discovered in the future, it can't work just like this; see discussion above about auto-gen of op types.]

Here's Maybe a Cute Idea

Instead of inventing yet another imperative language, allow op decls of the form

```
op(...)->(...) = {
    /* Normal C code, not including data type decls */
}
```

Note the use of curly braces to enclose the block, signaling C code is contained therein. We could even enforce large-grain functionalism, but disallowing global vars and pointers, which is pretty natural to do anyway.

Yet Again on Types versus Values

Is it in fact possible to stick with the idea that types and values are the same thing? Specifically, can we meaningfully reconcile the "obj is ..." construction (aka, abstract objects) with the "obj = ..." construction (aka, concrete objects)?

The closest we've seemed to be able to get to it is to think of concrete objects as a *restricted* type. E.g., with

```
obj Male = "male"
```

The type of Male is string, restricted to the specific value "male". At type checking time, this means that Male is treated exactly as if declared to be string. At runtime, it means that the only "male" can be bound to it. If an attempt is made to bind another string value to a variable of type Male, the variable will be set to nil.

On Injection Inference

It seems that this is possible when the elements of a union are unique, named types, otherwise it's not. Rather than restrict union elems arbitrarily/artificially to be named types, we'll just have the checker do its work silently, making injection inferences when it can, and complaining, possibly not explicitly to the point, when it cannot. For a bit of supportive human kindness, see Tennent page 215.

Extremely Hot News Flash (1 Dec 94)

That T2 is a subtype of T1 most emphatically does *NOT* mean that the value set of T2 is a subset of the value set of T1. In fact, it means the opposite. This concept is counter-intuitive when we consider the notion of subtyping integers and reals. E.g., with

```
obj subint < integer;
```

it is not the case that type subint is a subrange of integers. Rather, subint is a single-elem tuple type, the element of which is an integer.

More could definitely be said here, and it should be.

Back to Subtype Polymorphism, Yet Again

OK, maybe it's OK. (Make up your mind, clown!)

Screw Name Type Equiv

We don't need it any more. Forget mangling; we'll type check overloading by checking each def alternative.

Summary of Op Call Type Checking

This is a summary of the discussion below on type checking op calls.

- Foreach op of given name, check for parm match.
- Parm match defined as:
  - *o* If atype is same name as ftype, then OK
  - *o* If ftype is opaque,

Farg -- the above seems to be leading back to subtype polymorphism, which we think is bad. Let's try a purely equational definition to see what's up with all of this:

```
obj Thingy;

obj ThingyList
    ops:
                                  Insert(ThingyList,Thingy)->ThingyList;
                                  FindNth(ThingyList,integer)->Thingy;
end;
```

Now, is the following call possible?

```
Insert(1, Insert(2.5, Insert("xyz", null)));
```

Well, it just might be, and it might be type-safe as well, for the following reason. While the list can contain a bunch of different typed things, there's nothing that can be done with them unless we define some more ops on type Thingy.

OK then, let's try this:

```
obj Thingy
    ops:
        "_*_"(Thingy,Thingy)->Thingy;
end;
```

I.e., we've added a requirement to the definition of Thingy. Now, can we do the following?

```
FindNth(Insert(1, Insert(2.5, Insert("xyz", null))), 1) *
    FindNth(Insert(1, Insert(2.5, Insert("xyz", null))), 3)
```

!!!!!!!!This question must be answered!!!!!!!! We'll do so at our next meeting. Or at least try to do so.

Here's a crack a (partial) answer now -- the solution may lie in the *variable* nature of opaque types in an op signature. Viz., if there is > 1 opaque type in a signature, then as soon as the first parm of that type is bound, it *fixes* all other occurrences of that opaque type in the signature to be the initially-bound type.

OK, here is, I believe, the definitive answer on this subject. Viz., *there is NO subtype polymorphism allowed*, and, in fact, no genuine polymorphism at all. Rather, apparently polymorphic functions are obtained via the generic instantiation mechanism. See rsl3/tests/generic-list.rsl, about which more should be written here.

Overloading and Vararg Ops

Is it possible to combine overloading and the ML-style single-arg functions? Let's see. Consider the following definitions:

```
var x:X, y:Y;
obj X is int,real,boolean;
obj Y is string,boolean;
op F1(X)->(Y);
op F2(int,real,boolean)->(Y);
op F3(X)->(string,boolean);
op F4(int,real,boolean)->(string,boolean);
```

Are F1 through F4 all the same function type? More precisely, do all of the following calls type check?

```
F1(x,y);
F2(x,y);
F3(x,y);
F4(x,y);
F1(1,2.5,true);
F2(1,2.5,true);
F3(1,2.5,true);
F4(1,2.5,true);
```

Emerging rule: a function of a single arg can be called with unbundled actuals, but not vice versa. I.e., a function of multiple args cannot be called with a single arg. I.i.e, actuals will be bundled automatically when necessary, but not unbundled. Now we need to define the unbundling precisely. Also, we need to explain *why* this unbundling strategy has been chosen. And, it better be a good explanation. I.e., this strategy should only be chosen if it avoids some actual ambiguity or if it saves a significant about of conceptual and/or implementation difficulty.

```
obj class Any;
obj FullyGenericFunc is op(Any)->(Any);
```

Classes as Unions

They are. Use "?<" op to check for membership. But this is an old idea, now out of date. Say what in that last sentence??!? I do believe that "?<" and its ilk are alive and well.

Name Type Equivalence and Type Variables

In order to support type vars with name type equiv, the following rule seems reasonable:

> Are opaque types are distinct, unequivalent to any other type.

Precisely how this rule actually affects things needs still to be worked out. See the next section for further, hopefully clarifying discussion.

The Low-Down on Polymorphism and Name Type Equivalence

OK, let's try the following simple example:

```
obj Any is any;
obj AI instance of Any is integer;
obj AR instance of Any is real;

op w(a:Any)->() =
    WriteReal(a);                  (* Should not type check. *)

op e(a1:Any, a2:Any)

op AddRecord(gdb:GenericDB, gr:GenericRecord)->(gdb':GenericDB);
    (* *)
obj PRecord instance of GenericRecord;
obj SRecord instance of GenericRecord;

var gdb:GenericDB;
    pr: PRecord;
    sr: SRecord;
```

OK, try this rule on for size:

> An opaque type represents, among other things, a type variable.

Let us consider carefully the ramifications of this rule, particularly in comparison to the concept of representing type vars lexically, as in ML, with some notation such as a leading or trailing question mark. The disadvantage of opaque types as type vars is that it creates sort of a special case for type checking proc calls, in that the type of an actual corresponding to a type var can be any type for which equality is defined. This is really no different than an ML type var, except I'm still a bit uneasy about it somewhat.

The advantage of opaque types as type vars is its general orthogonality. In particular, constraints can be placed by the normal operations and equations attributes rather than by some special-purpose notation. E.g.,

```
op F(x:X?, y:Y?)
    where: exists G(X?,X?)->Y?;
```

versus

```
obj X
    ops:
                                  G(X,X)->Y;
end;
```

Now consider:

```
obj Some;

obj SomeAndSome is s1:Some and s2:Some;

obj SomeList is Some*
```

What exactly can be done with/to these objects, and what is the nature of the polymorphism enabled by considering obj Some to be a type var? E.g., is Selects1(SomeAndSome)->(Some) a polymorphic function for any type Some? Seems that things are OK, since SomeAndSome needs to be thrown in to any call.

Let's consider some operations:

```
op SomeOp(s1:Some, s2:Some)->(s1':Some, s2':Some);
op SomeOp(s1:integer, s2:Some)->(s1':Some, s2':Some);
```

Both of these ops can be invoked with a pair of ints. But, let us say, that the 2nd will cover the first, since it is more specific. I think this is a viable rule.

Let's see if we can (begin to) articulate the parm binding type checking rule:

1. If the function desig is a name, get the list of signatures available for that name.
2. If the function design is not a name, type check it to obtain the single signature designated.
3. Determine the types of each of the actuals, and look first for an exact match, probably via mangled name lookup.
4. Then look for a polymorphic match.
5. Finally, look for an exact or polymorphic match via bundling.

Somewhere in there we need to handle the chameleon functions -- those overloaded based on need. The best built-in example is [...] (but see revision below). It seems that this class may include in general functions overloaded on coarity only.

Chameleon Operations

For starters, see tests/deriving-types-from-obj-exprs.rsl for an example and a bit of discussion. Botton line of this test file is that the type of expr [c1,...,cn] is T* if typeof(ci) = T for all i, else the type is typeof(c1) and it could be interpreted as former if necessary in a given context.

In general, I believe that the definition of a "chameleon" expression is one which can be of two or more types, depending on the context. From a purely functional foundation, such chameleons derive solely from chameleon functions, which are defined as functions overloaded on coarity.

Unoverloading [...]

Major news -- let's nuke {...} as a comment delimiter and use it instead as ando constructor. This is largely in keeping with extant PLs, such as C. It also means that [...] is no longer a chameleon, which will no doubt simplify things. It also opens the door to an interim version 3 that does not allow user-defined chameleons.

On the Top-Level Module Context

Try this one on for size. In the past, we have considered hopping back and forth between modules at the top level in order to change contexts. As an alternative to this we could say that there is exactly on top-level scope, conceptually the module Main. The effect of switching scopes can be accomplished by opening and closing files, using the Open and Close environment ops, explained shortly.

On Modestly Intelligent Import/Export

Import/export rule summary:

a. Default for any module is to export all of its symbols, thereby making all of its symbols available for import.

b. If a module contains an explicit export declaration, then only those symbols exported are available for export. (This is a bit odd, but seems to fit best with the "say-the-least, get-the-most-expected" principle.)

c. "import x", where x is not a module, imports x unqualified if

    i. x is an export of exactly one opened module

    ii. x is not defined in the importing scope

d. "import x", where x is not a module, imports x qualified if

    i. x is an export of exactly one opened module

    ii. x *is* defined in the importing scope

    iii. (a warning is issued upon qualified import)

e. "import x", where x is not a module, fails with message if x is export of no module or more than one module.

f. "import m", where m is a module name, imports all of the exports of m, per the individual "x" import rules defined above.

g. There is an Open function that opens a file, and imports all of its exports into the current scope.

h. There is a Close function that unimports all the exports of a file.

Before any module can be imported from, in must be opened. Opening happens when a file containing one or more modules appears as a command-line arg, or when a module name is given as an argument to the Open environment function.

i. Only module, object, and operation names may appear in import/export decls.

j. Exports may not be third party. I.e., a module may not export any of its imports.

k. Import names may be qual idents, indicating from which module an import should come. Qual ident imports follow the same import rules defined above. In particular, an import as a qual ident does not automatically need to be qualified in its usages.

l. Given the two-pass nature of importing described below, imports will be processed *after* all defined module entities. Therefore, an entity definition will never be processed after an import, so that an error of the form "obj already def'd as import" will never occur.

Note that one of the effects of the behavior of Open is that all command-line files containing modules are opened and have their exports put in the Main scope. This means that when the environment starts up, the top-level contains all exports of all modules.

This seems to be pretty good. It means that the normal expectation of complete visibility happens automatically at the top level. At the same time, modules themselves do not automatically gain access to other modules symbols without explicitly importing those symbols.

Another effect of the preceding rules is import at the top level is basically meaningless, or at best redundant. This is the case because command-line files are automatically opened and opening in turn automatically imports into the current scope. To counter this, we might say that Open only makes symbols "available" for importing into the top-level, rather than automatically importing them. However, this seems like a bit of a nuisance, since the (vast) majority of the time, users will want to open and then import at the same time. The only possible advantage of not importing on open is that we could avoid conflicts by creating a top-level "pool" of modules with symbols available for import. Then by selective symbol import, we could grab things into the top level. This seems far less likely to happen, particularly since we're trying to deemphasize information hiding altogether at the specification level.

A low-level implementation observation is that import/export must (should) be accomplished in the second pass. Viz., the first pass will parse all modules, thereby making each of their exports available for import. The second pass will process all imports (which must lexically precede all other module decls), so that:

a. there is no forward ref problem with importing

b. imported symbols in a particular module will be processed at the beginning of the second pass, after which module type checking will precede.