Thorn: Draft Spec v0.5

Bard Bloom
John Field
Nathaniel Nystrom
Johan Östlund
Gregor Richards
Rok Strňisa
Jan Vitek
Tobias Wrigstad

Grammatical Conventions

In this document, I use *A ‡ B* for *A (B A)***;* that is, a list of *A*'s separated by *B*s.
A § B = [A ‡ B], an optional list of *A*s separated by *B*s.
A ¡ B means *A or B or both, in that order.*
A ¿ B means *A, B, neither, or both in either order.*
A? and *[A]* are equivalent notations for optional *A*.

Lexical Matters

Comments	Comments are delimited by any of: `//` and end-of-line `/*` and `*/` `#` and end-of-line Compiler Limitation: `**/` doesn't end a comment properly.
Identifiers	Either of the following - A letter, followed by by zero or more letters, numbers, `.`'s, `?`'s, `!`'s, and `_`'s. - Any characters (except for the copyright character ©) enclosed in backquotes, except for newlines and backquotes and stuff which breaks JavaCC. - Thorn keywords can only be made into variables with the second (backquoted) form. - The copyright character © is not allowed in programmer-defined identifiers; identifiers containing it are reserved for the system. Case *is* significant in identifiers. Latin-1 Supplement letters and Latin Extended A are currently allowed in identifiers.
Strings	Characters (not newline) enclosed in either single or double quotes; `\` escapes single and double quotes, `$`, and itself, and the escapes `\n \r \t \b` work too. Also, characters enclosed in either kind of quote triplicated work as well. No escapes apply in that context. **Note**: Strings allow variable interpolation. That is: in a double-quoted string, `$x` and ``$`x` `` are both replaced by the value of variable `x` (converted to a string by the `str` method)
Floats	The syntax for floats is *Digits* `.` *Digits [[*`e`|`E`*] [-] Digits]*.
Ints	*Digits* `0x` prefix for hexadecimal `0o` prefix for octal `0b` prefix for binary
Semicolons	Semicolons *terminate* statements that don't have an obvious terminator otherwise. Some control constructs that end with a close-brace '`}`' allow trailing semicolons, and some don't. (This is a parser issue, and we are working on it slowly.)

Common Stuff

Actuals	`(` *Exp § , [,]*`)`
QualName	*Id ‡* `.`
BlockOrExp	*Block | Exp*
PurityTest	*[* `: pure` *]*
TypeConstraint	*Id*
TypeConstraints	`:` *(TypeConstraint ‡* `&`*)*

Closing Brackets

In most cases, closing brackets can have an optional tag telling what is being closed. The tag has the form `}keyword` or `}id`. Commands that have a keyword can be closed with the command's keyword. Commands that declare an identifier can be closed with the identifier being declared. Commands that do both allow both closing markers. In no case is the closing marker required.

-
  ```
  try {

    x := y;

  }try

  catch {

    a = d()

  }catch
  finally {
    e(f);

  }finally
  ```

-
  ```
  class C{
    def p(){ 1; }p
    def q(){}def
  }class
  ```

Classes

ClassDecl	`class` *Id ClsFormalList? PurityTest ClsExtends ? ClsBody*
ClsFormalList	`(` *ClsFormal § ,* `)`
ClsFormal	*Formal*
	`var` *Id*
ClsExtends	`extends` *ClassDeclSupercall ‡* `,`
ClassDeclSuperCall	*QualName* `(` *Exp §* `,)`

ClsBody	`{` *ClassMember* `}`
	 `;`

A small example:

```
class Cplx(x,y) {
  def r() = sqrt(x*x + y*y);
  def theta() = atan2(y,x);
  def str() = "$x+i$y";
}
```

The formals provide a constructor, field definitions, and a pattern, thus:

- Every variable in the formals but not in the ClassDeclSupercalls is a `val` field of the class.
- The default constructor (a) calls the super-constructors given in the ClassDeclSupercalls, and (b) binds the remaining variables locally. If the class defines a constructor of the same arity, no default constructor is provided.
- The default extractor extracts the fields named as the class parameters. If the class defines an extractor of the same name, no default extractor is provided.
- **Limitation:** as of now, class formals must be variables (not more complex patterns), and the actuals passed to superclasses in the ClsExtends clause must be variables or constants. This is not an inherent problem, just a cost-saving measure.
- If a class formal is forwarded to a superclass, it must be declared as just an identifier – no `var` or type.

For example:

```
class C(a,b,var c) extends D(a) {}
```

amounts to

```
class C{
  val b;
  var c;
```

```
  new C(a',b',c') { new@D(a'); b=b'; c:=c'; }
}
```
except that the former also provides a pattern extractor, which the latter does not.

Assorted Notes:

- If the class body is empty, the `{}` can be replaced by a `;`, thereby saving one character.

Pure classes (those defined with **`:pure`** in the header) have additional restrictions and features:

- They cannot have or inherit **`var`** fields.
- They cannot refer to external **`val`** or **`var`** fields.
- All their fields must be hereditarily immutable (determined at runtime).
- They automatically get an == method defined by type and field equality, unless you provide something else.
- Their instances are transmissible. No other instances are transmissible.

Restrictions on Multiple Inheritance:

- **TO DO**

Class Members

ObjMember	*FieldDef*
	MethDef
	MatcherDef
	Import
ClassMember	*ObjMember*
	CtorDef

Field Declarations

FieldDef	**`val`***? Pat = Exp ;*
	`val`*? Id;*
	`var` *Id [:= Exp ;]*

Fields can be defined by the same constructs that define local variables, plus one more to declare immutable fields that will be initialized by the constructor.

```
  var C_counter := 0;

  class C {
    which_C = ({ C_counter += 1; });
    var w := 18;
    val v;
    new C(v') { v = v'; }
  }C
```

`val` fields which don't provide an initializer are allowed as fields. (They are *not* allowed as bindings in blocks.) Fields defined in bodiless **`val`**s must be bound by constructors.

The treatment of instance variables is this. The instance variables defined in a class are visible from that class (**not** its subclasses – a change from previous versions of Thorn), but only for **`this`**. Thorn provides default getters and setters for all instance variables, unless the user has written different ones. From outside, all access to fields is done by method call. The syntax **`x.f`** is simply an abbreviation for the nullary method call **`x.f()`** and **`x.f := g;`** is sugar for **`` x.`f:=`(g); ``**.

The class **`A`** below, from outside, seems to have var fields including **`b`**, **`c`**, and **`e,`** and val field **`d`**. As you might hope, **`a.b := a.b+1;`** will increment the **`b`** field of **`A`** instance **`a`**. **`a.c := 3;`** will work, but **`a.c := 4`** will fail, as assignment to the **`c`** field is specialized to only work for primes. There is no instance variable named **`e`** but **`A`**'s behave from the outside as if there were one: **`a.e := a.e+1`**; does just what one would expect. Attempts to access the representation variable **`not_e`** will always throw exceptions.

```
class A {
  var b;
  // implied getter: def b() = b;
  // implied setter: def `b:=`(b') { b:= b'; }
  var c;
  def `c:=`(c') {
     if (prime?(c') c:=c'; else throw "Please don't"; }
  // implied getter: def c() = c;
  val d = 1;
  // implied getter: def d() = d;
  // no 'e'
  var not_e;
  def e() = not_e;
  def `e:=`(e') { not_e := e';}
  def not_e() {throw "Please don't";}
  def `not_e:=`(x) {throw "Please don't";}
  }
a = A();
```

Method and Function Declarations

FunDef	**fun** *Id Formals PurityTest FunBody1*
	fun *FunCases*
MethDef	**def** *Id Formals PurityTest FunBody1*
	def *Id FunCases*
FunCases	*FunCase ‡* **and**
FunCase	*Id Formals PurityTest DistStuff FunBody1*
FunBody1	*Block*
	= *Exp* **;**
	= *Block*
Formals	*Pat*
	(*Pat §* **,** *[,]***)**
	-- an omitted formal list is considered **()**
DistStuff	*[***from** *Pat] [***envelope** *Id] [***prio** *Int]*

Methods are defined using **def**; Functions are defined using **fun**. Otherwise, the two declarations are identical syntactically.

```
fun square(x) = x*x;
fun twice(f,x) = {f(x); f(x);}
fun chant(msg, n) {
   for(i <- 1..n) println(msg);
}chant
fun car [x,_...] = x;
fun cube x = x*square(x);
fun three() = "tri";
```

1. Many methods in the same class may have the same name, as long as their arities differ. This is not true of functions. A **fun** definition introduces a new name.
2. Method calls *always* use method call syntax: **x.m(y)**. Function calls *always* use function call syntax: **f(y)**. In particular, an object must use **this.m(y)** to call its own method.
3. **from**, **envelope,** and **prio** are not allowed for **fun** or **def**, but are allowed and useful for the distributed **sync** and **async.** They are presented here purely to share the syntax.
4. **Convention:** Methods and functions which are intended as pure boolean test – that is, return true or false, and don't change any state – should have names ending in **?**, and nothing else should. This convention is not enforced.
5. **Convention:** Methods and functions which change state should either have names which unambiguously indicate change (*e.g.*, **add**, **incrementCounter**) or end in **!**. For example, a function **count(stuff)** can be expected to return the count of stuff. A method **counter.count!()** can be expected to increase the counter.
6. A method may have the same name as another class member. (This is not ambiguous: methods have arities and are used via method call syntax, unlike anything else that can appear in a class.) **Convention:** A nullary method with the same name as a field is a getter for the field; in the following example, **m()** is the conventional getter for **m**. Indeed, such methods are generated by default.

```
class A {
    var m := 1;
    def m = m;
  }
```

Constructors

CtorDef	**new** *Id FormalList FunBody*
CtorSuperCall	**new @** *QualName Actuals*
CtorThisCall	**new** *Actuals*

```
class C extends A, B{
  val see; // Ctor must init this.
  new C(x,y) {
    new@A(x);
    new@B(y);
    see = 1; // Binds the field of 'this'.
    }
}class
```

Other Notes:

- ☒ Uninitialized vals (`see` in the above) can be initialized in the constructor; the initialization must be at the top level of the `new`. The interpreter does not enforce this.
- ☒ `new` can be either a supercall (with the `@A` suffix) or a this-call (without it). This is different from `super`, which is always a supercall even without the syntax. **Rationale:** Constructors use `new`. `super` handles super-calls. This should not be remarkable.
- ☒ The object itself (`this`) also cannot be accessed in the constructor.
- ☒ The construction process calls the method `init()` on the newly-constructed object.
- ☒ If no constructor is specified, `C()` calls the nullary constructors of all superclasses.
- ☒ Fields are initialized in the order they appear in the class. Fields are initialized *before* constructors are called.

Anonymous Objects

AnonObject	`object` *PurityTest ClsExtends? ObjBody*

The *AnonObject* construct creates a single object, as if the `object` construct were a `class` declaration followed by a constructor call to that class.
Anonymous objects can have everything that classes can, except constructors.

```
object extends This,That {
   var a := 0;
   b = "const";
   def bump() { a := a+1; a; }
}
```

Import Declarations and Module Files

ImportAModule	`import` *[Id =]Id*`;`
ImportAMember	`import` *[Id =]Id.Id*`;`
ImportOwnModule	`import own` *[Id =]Id*`;`
ImportStar	`import` *Id*`.*;`
Import	*ImportAModule* | *ImportAMember* | *ImportOwnModule* | *ImportStar*

Import stuff from a module.
To Do: Document this. Especially `own`.

Future Work: `own` is deprecated and will be destroyed as soon as we can manage to do it.

Temporary restriction: Module imports must be acyclic.

Module Files

ModuleFile	`module` *Id* `{` *(ModuleBit)** `}`
	`module` *Id* `;` *(ModuleBit)* EOF*
ModuleBit	
	Import
	ClsDecl
	FunDecl
	Bind
	VarDecl
ModuleMember	*Visibility*
	Alias
FileName	*StringLiteral*
Visibility	(`public` | `private`) *Id*

```
module ID {
   var idcounter := 0;
   class Id {
     val id;
     new Id(){
       idcounter += 1;
       id = idcounter;
     }new
   }Id
}ID
```

Expressions and Statements

Stmts	*Stmt* *
Exp	*Stmt*
Stmt	*IfStmt*
	LabelledLoop
	Try
	Match
	Break
	Continue
	CondExp
	FunDef
	ObjectDef
	Import
	Block
	MapPut
	Binding
	Assignment
	Return
	Throw
	ImportStmt
	Exp `;`
	CtorSuperCall
	CtorThisCall
	TableStmt
	RecordStmt
	ComponentStmt
	SucceedStmt
	FailStmt
Block	`{` *Stmts* `}`
Exp	*ClosureExp*
	Call
	Exp Binop Exp
	Unop Exp
	FieldRef
	Query
	TableExp
	RecordExp
	ComponentExp
	Literal
	ConditionalExp
	CollectedExp
	MapGet
	MapCtor
Cmd	*Stmt* | *Exp* --- This is mostly an AST concept. Statements and expressions are syntactically different, but semantically identical. Indeed, there are conversion syntax allowing either to serve as the other.

1. *Exp* is a single expression – or equivalently a single statement. An *Exp* is good for, say, an argument to a function, or either side of an assignment.
2. A *Stmt* can stand by itself in sequential code; it doesn't need (and sometimes, annoyingly, can't have) a trailing `;`. An *Exp* followed by a semicolon is one kind of *Stmt*.
3. Some *Stmt*s can be put in parentheses to make expressions: *If, Try, Match, Recv, Throw, Block*. This allows:

```
a = ({x = 10; x*x + 1;});

b = (match([1,2]) {[x,y] => x+y});

c = (if (true) 3;);

    // equivalent to: 3 if true else null;
d = (try{ throw "ow!"; } catch { x => x });

e = (recv{x from $(kim) => x timeout(1000) {null;}});
f = ({i += 1; j -= 1; [i,j];});
g = 1 if n>0 else (throw "Negative?!";);
```

Bindings and Assignments

Binding	*[*`val`*] Pattern = Exp*
Assignment	*Exp ‡* `,` `:=` *Exp ‡* `,`
	Exp (`+=` | `-=` | `*=` | `/=` *) Exp*
VarDecl	`var` *Id [TypeConstraints] [*`:=` *Exp]* ;

For parsing reasons, bindings and assignments are expressions. They usually appear as statements though, followed by a semicolon.

Bindings do pattern matches. If the match fails, the binding throws an exception. If it succeeds, it binds all the variables from the pattern for the rest of the current scope:

```
        a = [1,2,3];
        [b, c, d] = a;
        // c == 2, etc.
```

Binding is most often used with a single variable, `x=1;`.

Assignments assign to Thorn's small collection of assignable locations: `var`-defined variables (`x`), fields of objects (`a.f` – and that is really a method call), and rows of tables (`T(x)`, `M[y]`). Assignments are done in parallel, so `x,y := y,x` swaps `x` and `y`. The left and right side of the assignment must have the same length.

```
a = 1;
[b,c] = list2();
x := x+1;
x,y := y,x;
x.f := 3;
T(1) := {: v: 2 :};
```

`T(x) := y` is an abbreviation for `` T.`():=`(x,y) ``.

Note: `=` and `:=` are not simply variations on the same concept. `=` introduces new variables; `:=` operates on existing variables.
Assignment to a pattern is not allowed.
`/***/ [b,c] := list2();`
Nor is rebinding variables : `/***/ x,y = y,x;` is not meaningful in Thorn.

If a type constraint is placed on a variable, every assignment to that variable is checked (statically or dynamically). It is an error if any non-conforming value is ever assigned to that variable.

If/Unless

If	*Maybe* `(` *Exp* `)` *Stmt* *[* `else` *Stmt]*
Maybe	`if` | `unless`

There is no particular else-if construct, but it is not needed:

```
  if (x == 1) y := "one";
  else if (x == 2) y := "two";
  else y := "huh?";
```

Top-level conjunctions of pattern matches and other expressions have a special status with respect to `if`: bindings introduced by the successful pattern matches are available in the true-branch of the conditional (*viz.*, the then-branch for `if` and the else-branch for `unless`.) For example, the following binds `a, b, c` in the then-branch:

```
  if (f(x) ~ [a,b,c] && a>b>c ) {y := a+b+c;}
```

If without an else-branch returns null.

Loops

LabelledLoop	*[Id*`:` *] Loop*
Loop	*While* | *DoWhile* | *For*
While	*(* `while` | `until` *)* `(` *Exp* `)` *StmtOrBlock*
DoWhile	`do` *Block (* `while` | `until` *)* `(` *Exp* `)`
For	`for` `(`*Pat* `<-` *Exp*`)` *StmtOrBlock*
	`for` `(`*Pat* `<~` *Exp*`)` *StmtOrBlock*
Break	`break` *Id?* `;`
Continue	`continue` *Id?*`;`

Unbounded loops are pretty conventional, except concerning bindings:

```
while(p(x)) { x := x+1; }
until(p(x)) { x := x+1; }
do { x := x+1; } while (p(x))
do { x := x+1; } until (p(x))
```

Bounded loops are iterations over lists (or other iterable structures). They aren't very C/Java like.

```
for (i <- 1..10) tense(i);
```

All forms of loops allow labels, and C-style `break` and `continue` out of multiple levels of loops.

```
seek: for(i <- 1..10) {
  for([j,k] <- L)
    if (t(i,j,k)) {found := true; break seek;}
}for
```

Note that the binding in a `for` includes a pattern match. The two variants of `for` differ in the treatment of matching:

- `for` with `<-` is **demanding**. Every element of the collection must match the pattern; if it does not, `for` throws an exception. This behavior is like `=` for binding. It should be used when you know what structures you're iterating over, and will be upset if the things in the structure don't match it.
    - `for( i:int <- [1,2,'boom'] )` will throw an exception when it gets to the non-integer.
    - `for ( {: k:3, v :} <- amap )` iterates through all elements of `amap` with `k` equal to `3`, exploding if any element has a different key. If `amap` is a map produced by `map()`, this loop will fail unless `amap` is empty, or has just the one binding for `3`.
- `for` with `<~` is **inquisitive**, like the `~` operation. If the pattern doesn't match, it is skipped, without an error. `<~` is useful when you are looking for something (or for several things), especially through a heterogeneous collection.
    - `for ( i:int <~ [1,2,'boom'] )` will evaluate its body twice, for `1` and `2`; it skips `'boom'`.
    - `for ( {: k:3, v :} <~ amap )` iterates through all elements of `amap` with `k` equal to `3`. If `amap` is a map produced by `map()`, the body will be executed once if there is a value for key `3`, and not at all if there is not.

Bindings in the test of a `while` loop are available in the body.
Bindings in the test of an `until` or `do-until` loop are available after the loop.
Bindings in the test of a `do-while` loop aren't available anywhere.

Pattern Matching Statements

Match	`match ( `*Exp*` ) { `*Cases*` }`
Cases	`|`*? (Case ‡* `|`)*
Case	*Pattern From? Prio? [* `=` *|* `=>` *] (Exp | Block)*
From	`from` *Pattern*
Prio	`prio` *Exp*

`match` selects the first clause for which the value of *Exp* matches the pattern, and evaluates and returns the corresponding *Exp* or *Block.*
`from` clauses are matched against the sender in `receive`; they are forbidden in `match.`
`prio` clauses give message priority in `receive`. In `match`, they have no semantic effect, but higher-priority clauses must precede lower-priority; this is intended to document that certain clauses are supposed to be before other ones. If no priority is given, the priority is 0.
There is no difference between `=` and `=>` in matching statements.

```
match(x) {
    1 => "one"
  | [z] => "singleton"
  | [z, ($(z)) ... ] => "repeater"
  | _ => {
      mysteries @= x;
      "mystery";
      }
  }match
```

Patterns in Thorn

Id	`x`	Matches any subject; binds that identifier to the value matched.

Literal	`1`
	`"two"`
		Matches that value

`$`*Id*
`$(`*Exp*`)`
	`$x`
	`$(x+1)`
		Evaluates the identifier or expression; matches its value.

`_`	`_`	Wildcard; matches anything; binds nothing.

`[` *(Pat* `...`*?) §* `,` `]`	`[]`
	`[1]`
	`[1,x...,2]`
	`[x,y...]`
	`[[y,z]...]`
		Matches a list. Within that list, a *Pat* matches a single element and binds everything that that match entails;
		Pat`...` matches a list of zero or more things, each of which matches the pattern. So `[x,y...]` gets the car and cdr of a list. Ellipses in the middle of patterns imply backtracking. (Pattern matching doesn't do any other backtracking).
		`[1,1,1,2,3,4] ~ [a..., 1, 2, b...]`
		succeeds, with
		a = [1,1]
		b = [3,4]
		The subpattern `x...` matches a sublist and binds it to `x`. The subpattern `(x && P)...` matches a sublist, each of whose elements match `P`, and binds it to `x`. `P...` matches a list of elements each of which match `P`, but does not bind it.
		No other patterns do binding – they are all elementwise tests (and in fact bind their vars to the *last* value they had, which is Wrong.)

`{:` *(Id*`:` *Pat)*
§ `,` `:}`
	`{:`
	`a:x,b:y :}`
	`‹a,b:b0›`
		Matches a record or object whose fields match the given pattern, and may include other fields.
		TODO: Syntax for an "and no other fields" clause.
		Also, `x` abbreviates `x:x`

`it`	`it`	**Expression, not pattern.** Can only appear in patterns. Its value is the subject of the current pattern. For example, `[(it > 3)?...]` matches a list of numbers each of which is greater than three.
`+`*Pat*	`+x`	Fails if the subject is null; otherwise undoes one unary `+` operation and matches the result against *Pat*. See discussion of the `null/+x` idiom in the paper.
`(` *Pat* `)`	`(x)`	Matches whatever the pattern does. Just parentheses.

Pat1 `&&` *Pat2*	`x && [y,z]`
	
	`[_..., 1,`
	`_...] &&`
	`[_... 2,`
	`_...]`
	
	`[x,y] &&`
	`(x>y)?`
		Matches things which match both *Pat1* and *Pat2*, producing all the bindings that either one produces. (Bindings from *Pat1* are available in *Pat2*).
		☒ `x && P` binds the subject to `x`, giving the effect of ML's `as`.
		☒ `[_..., 1, _...] && [_... 2, _...]` matches a list which has both 1 and 2 as elements, in either order.
		☒ `[x,y] && (x>y)?` matches a two-element list whose first element is larger than the second. This idiom obviates the need for the `where` clause of some pattern-matching systems; in Thorn, we may use `&&(`***`cond`***`)?` to insinuate a side condition ***`cond`*** at any point in a pattern match. In particular such side conditions can stop matching partway, which `where`-style side conditions cannot.

Pat1 `||` *Pat2*	`[_] ||`
	`[_,_,_]`
		Matches anything that either *Pat1* or *Pat2* does. This never produces any bindings. `[_] || [_,_,_]` matches a one- or three-element list.

`!` *Pat*	`!(_:int)`
	`!([x, $x])`
		Tries to match the subject against *Pat*. If that match succeeds, `!` *Pat* fails. If that match fails, `!` *Pat* succeeds. In either case, `!` *Pat* produces no bindings. `!(_:int)` thus matches any non-integer. Bindings inside of *Pat* behave normally; thus `!([x, $x])` matches anything but a list of two identical elements, as `x` is bound inside of *Pat* but not outside.

`(` *Exp* `)?`	`( f(it) >`
	`g(it) )?`
		Evaluates *Exp*. Passes if the result is true; fails if it is false; is an error otherwise. Never binds anything, even if *Exp* produces bindings.

Exp ~ *Pat*	`f(it) ~ []`	Evaluate a pattern match on a different subject, *viz.*, the value of *Exp*.
Pat `..` *Pat*	`x .. y`	Match a range.
Pat `:` *Id*	`x : int`	Type test. Matches if the subject has the type given by *Id* (or a subclass if applicable), and matches *Pat.*
Id `(` *Pat §* `,` `)`	`Cplx(x,y)`	Call an extractor on the subject. Extractors are the inverse of class default parameters, only.
`.`*Id*`(`*Pat*`)`	`.nice([x])`	Abbreviation for `it.`*Id*`() ~ +`*Pat*
`.`*Id*`()`	`.nice?()`	Fails if `it.`*Id()* returns `null` or `false`; succeeds otherwise. Abbreviation for `it.`*Id*`() ~ !(null || false)`

Exp `/` *Pat*	`"(a*)(b*)" / [a's, b's]`	Abbreviation for `` it.`match/`( ``*Exp*`) ~ +`*Pat*. Typical usage is with the *Exp* a string holding a regexp, and the *Pat* a list pattern to match the substrings from the groups appearing in the regexp. The example matches a string of a's followed by one of b's, putting the a's in `a's` and the b's in `b's`.

Return

Return	`return` *[Exp]* `;`

Your basic return statement. We don't have a concept of subroutine that doesn't return a value, so if *Exp* is omitted, this returns null. Note that `return` is *not* required; subprograms return the last expression in their block normally. Thus the following are equivalent:

```
fun f(x) = x+1;
fun f(x) { return x+1;}
```

However, `return` can clarify complex code.

Exceptions

Try	`try` *Block (Catch ¡ Finally)*
Catch	`catch {` *Cases* `}`
Finally	`finally` *Block*
Throw	`throw` *Exp*

This is like Java in many ways. Exception values can be anything, though; there is currently no hierarchy of exceptions.

```
try {
     x := 1;
 }
catch {
     "Doom!" => {print("doom");}
   | "Mood?" => {}
   }
finally {
     x := 2;
   }
```

Call

Call	*Exp . Id Actuals*
	Exp.Id
	`super @` *QualName Actuals*
	`super .` *Id Actuals*
	Exp Actuals
FieldRef	*Exp . Id*

Pretty standard stuff. However, the supercall that specifies which parent class (and it must be a parent class) has slightly peculiar syntax. `super@A.B.C.D.methu(45)` calls the `methu` method from parent class `A.B.C.D`. Both the parent class and the method name must be specified, so `super@a(x)` is not a legal supercall. `super.f(x)` is fine, for a class that has only a single parent class.

`x.f` is a synonym for `x.f()`. Various other syntactic forms are also synonyms for method calls; *e.g.*, `x+y` is a synonym for `` x.`+`(y) ``.

```
x.methy(41),
x.fleld,
funcy(arg,arr),
super.methu(43),
super@A.B.C.D.methu(45)
```

Closures

ClosureExp	`fn` *Formals* `=` *BlockOrExp*

	`fn` *Id* = *BlockOrExp*
	`fn` = *BlockOrExp*
	`fn` *FunCases*

```
a = fn(x,y) = x*5;
b = fn x = x*x*x;
c = fn(x,y) { a := 1; b := 2; };
d = fn 1 = 0 | 2 = "zero";
```

Actually the body expression is a *CompactExp*, which is medium-high in the *Exp* hierarchy as implemented. In particular you can't have `fn = fn = 1.` The issue here is the multi-case function: `fn 1 = fn 2 = too | fn 3 = three` is ambiguous. Explicit parentheses save the day in the case of nested functions.

Lists

ListConstructor	`[` *ListBit § , [,]*`]`
ListBit	*Exp* *Exp* `...`
ListConstructorPattern	`[` *(Pattern ...?) § , [,]*`]`

The list constructor constructs a list, with the *Exp*s as elements, and the *Exp*`...`s as sublists. `[x...,y,z...]` is a list whose elements are those of `x` at the beginning, then `y` itself, then the elements of `z`. This could all be done with cons (`::`), nil (`[]`), and append (`@`), but sometimes the constructor is slicker. The list constructor pattern is approximately dual to the list constructor, so that

```
  fn [x,y...,z] = [x,y...,z]
```

is an identity function on lists with at least two elements.

```
[],
[1],
[1,x...],
[x...,y,z...]
```

There can be a comma after the last element in the list:

```
x = [Foo(1),
     Foo(2),
     Foo(3),
    ];
```

Tables

The table constructor has syntax:

TableCtor	`table (` *TableKey ‡* `, ) {` *TblColDecls* `}`

TblColDecl	*[*`val`*] Id TypeConstraints* `;`
	`var` *Id TypeConstraints* `;`
	`map` *[* `var` *|* `val` *] Id TypeConstraints;*

TableKey	*Id TypeConstraints*

For example

```
      t1 = table(userid){ homeDir; var usage; };
      t2 = table(x,y){var fg; val bg;}
```

The fields in parentheses are the keys. Their order is important, and used in many operations. The fields in braces are non-keys. Their order is not important.

The type constraints are checked when rows are inserted or mutated.

1. It is an error to try to put a non-type-conformant value into the table in any way.
2. It is not an error to try to subscript the table by a non-conformant key, though by the table's type invariant that key won't be there.

Inserting and modifying records of the table uses `:=`, thus:

```
      t1("kim") := {: homeDir: "/u/kim", usage: 0 :};
      t2(0,0) := {: fg: null; bg: greenSquare :};
      ...
      t2(0,0) := {: fg: player; bg: greenSquare :};
```

The record on the right needs to have all the non-key fields of the table listed. It may have other fields (including key fields), but they are ignored.

Deleting a record is done by `:= null`, thus:

`t1("kim") := null;` // bye-bye kim

Individual rows can be retrieved by subscripting with `()`:

```
r = t1("pat");
if (r ~ {: homeDir :} && ! homeDir.exists? ) {
      homeDir.mkdir!;
      }
```
This returns `null` if the subscript isn't a row of the table. The usual idioms for lookup apply here:
```
if (t1(usr) ~ +_) { /* usr is there */ }
if (t2(x,y) ~ {: bg, fg: null :}) {paint(x,y,bg);}
else if (t2(x,y) ~ {: fg :}) {paint(x,y,fg);}
```

If you're sure that a row exists, you can manipulate it by field selection:
```
t1(usr) := {: homeDir: hd, usage: 0:};
t1(usr).usage += 1;
```

Keys
Only certain kinds of things can be keys. Keys need to have a (low-level) concept of hashing and equality, which the table algorithms require. Keys must be

- Built-in values, and, if composite, all child values must also be keys.
- Objects of user-defined classes which have == and `hashCode()` defined and behaving properly.

Mapform Tables
A table may have one field ***`mf`*** flagged with `map`. This field can be accessed by square brackets. In most respects, `t[k1]` behaves like `t(k1).`***`mf`***.
```
dict = table(word){ map meaning; var count; };

print( dict["bailiwick"] );
```

The exception is assignment. Assigning to `t[k1]` if it's in the table is like assigning to `t(k1).`***`mf`***. However, if `k1` *isn't* in the table, assigning to `t[k1]` creates a new row with all other fields set to null.

```
dict.clear();
dict["syzygy"] :=
    " any two related things, either alike or opposite.";
assert( dict("syzygy")
   ~ {: meaning: _:string, count: null :});
```

When one is changing code by adding columns to a map, map assignments are places where bugs can occur. One may wish to have the invariant that `count` is always a number, for example. We provide a flag so that map assignments throw exceptions, to make it easier to locate map assignments that need to be updated to table row assignments.

Also, ordinarily `m[k]` returns `null` if `k` is not a key of `m` – and, of course, it also returns `null` if `k` *is* a key of `m` and has the map field set to `null`. The usual table operations suffice to branch on whether `k` is a key or not: `if( m(k) ~ {: v :})` succeeds and binds the map field (if it's called `v`). However, there is a flag to cause `m[k]` to throw an exception in this case.

`forbidMapSet(f)`	If `f` is true, the map assignment `t[k] := v` will throw an exception.
`forbidInvalidMapAccess(f)`	If `f` is true, invalid map accesses `m[k]` (when `k` is not a key) throw an exception rather than returning null.

`map()` abbreviates `table(k){map var v;}`. `k` and `v` are literally those identifiers.

Table Operations
Iterating over a table iterates over its rows, in unspecified order.

Some table operations mirror the operation-defined ones, but work with records – either records that look like whole rows, or records that look like keys, as appropriate. Some of these operations are special cases of the basic operations: *e.g.*, you may want to know that you're inserting a new row rather than updating an extant one.

Tables are equal (==) iff they they have the same set of column names and are equal considered as sets of records. This has some surprising consequences – for example, `table(a){val b;} == table(b){val a;}`. All the other definitions have surprising consequences too. (To check whether two tables are the same, use the `same?()` predicate)

Ordered Tables.

OrdTableCtor	`ord {` *TblColDecls* `}`
	`ord()`

`ord()` abbreviates `ord{map var v;}`.

An ordered table is like a cross between a table and a list. It has some elements, in sequence. Each element has fields, as for a table.

- Indexing is by integers, as for a list. Position 0 is the start of the list.
- Negative integers count from the end of the list.
- One field may be singled out by the `map` keyword. This element can be accessed by `[n]`-style subscripting. As with tables, `(n)`-style subscripting gets the whole row, as a record.
- Subscripting by a negative number counts from the end of the list: `-1` is the last element.
- Iterating over an ord iterates over the map field. If you want to iterate over the rows, use `for(r <- o.rows())`.
- Type constraints work just as for tables.
- `o @= v` appends a new value to an `ord()`. `o.add(r)` adds a whole record. There are plenty of other operations; see the library listing.

Records

RecordCtor	`{:` *RecordField* *§* `, :}`
	NOTE: ‹ ... › is a synonym for `{: ... :}`. On the Mac, that's shift-option-3 and shift-option-4.
RecordExp	*RecordCtor*
RecordField	*Id* `:` *Exp*
	Id

`{: a:x, b:y+1 :}` constructs a two-field record.
Records are their own data structure, independent of all others.
Note that an identifier `x` alone in a record is an abbreviation for `x:x`

Operations

BinOp	`+`
	`-`
	`*`
	`/` -- floating-point division
	`÷` | `div` – integer division
	`mod`
	`&&` | `||` | `^^` ---
	`<, <=, >, >=, ==, !=, :?, in` --- not strictly binary; see below.
	`..` – range builders.
	`@` is list append.
	`::` is cons.
	`:?` is type test.
	`:` is type constraint.
UnOp	`!, not` – synonymous, boolean negation
	`-` --- negative
TO DO	
	No built-in meaning, and interpreted as method calls
	Unary: `@! @~ @$`
	Binary: `@+ @* @/ @-` (same precedence and rules as /)
	Ternary: `<@ @>      [@ @]`
	No built-in meaning, and interpreted as function calls
	Unary: `!@ ~@ $@`
	Binary: `+@ *@ /@ -@`

	Ternary: `<@ @>` `[@ @]`

- **TODO:** Explain precedence.
- Unary `+` is the inverse of the + pattern. For most values x, +x == x. +null, ++null, etc are distinct values (the “nullities”). For all x, +x ~ +y, and results in binding y to x. Also, +x != null.
- `+(v)` always matches the pattern `+x`.
- `÷` and `div` are truncating integer division.
- `/` is floating point division, always.
- `x : t` evaluates to `x`’s value if it is of type `t`, or an exception otherwise.
- `mod` is modulus. (`a mod b)` **TODO** needs to be defined sensibly in case of negatives or non-integers. `i mod (j .. k))`, for integers `i, j, k`, computes the unique element of `j..k` which is congruent to `i` modulo the size of the interval. Thus `i mod (0.. n-1)` is a good index into an `n`-element zero-based structure, and `i mod (1..n)` is a good index into an `n`-element one-based structure.
- `*` and `/` aren’t arbitrarily composable. `a*b/c` is fine. `a*b/c*d` and `a*b/c/d` is a syntax error, on the grounds that it’s not 100% clear what is intended. Use parentheses.
- `&&` and `||` and `^^` aren’t composable at all. An expression can only have one of the three (though of course it can have parenthetical subexpressions containing the others).
- `^^` isn’t really a binary operator; it’s an n-ary operator, and true when one of its n arguments is true. (This is *not* iterated xor! A xor B xor C xor D is true when an odd number of {A,B,C,D} is true, which is rarely useful. `A ^^ B ^^ C ^^D` is true when precisely one of its arguments is true.) `^^` *is* short-circuiting to the extent possible: `true ^^ true ^^ boom()` will return false and not evaluate `boom()`.
- Comparison operations can be chained. The chaining is pointwise: `a < b > c` is allowed, and equivalent to `(a < b) && (b > c)`. (If `b` were an expression with side effects, this equivalence would be false; terms in a chained comparison are evaluated only once.)
- `e :? C` tests whether `e` is an instance of class `C`. It returns true or false.
- `e in C` tests whether `e` is an element of list or range `C`. For other types, it calls `C.revIn(e)`. (This is not like most other operations, which ask their first argument; but collections are more likely to understand membership than arbitrary values are.)
- `a .. b` returns a *range*: a compact immutable data structure representing the integers from `a` to `b` inclusive.
- `+` adds numbers, concatenates strings, and otherwise invokes the `+` method.
- Generally operations are sugar for method calls to their left argument: `a+b` is ``a.`+`(b)``. Some aren’t: `&&` and `||`, and, depending on definitions, comparisons and test (because of being chainable.).
- Assigning operators (`x += 1`) return the new value stored in the LHS.

Types

The following types are available for type tests and such.

`int`	Equivalent to Java’s `long`
`float`	Equivalent to Java’s `double`
`bool`	Boolean
`string`	Strings.
`record`	Records
`table`	Tables
`list`	Lists
`range`	Ranges
`ord`	Ords
`file`	File handles
`dir`	Directory handles
`nullity`	+null, ++null, etc.
`fn`	Anonymous functions
`pure`	Pure values
`bytes`	Byte arrays
`char`	Characters

To Do: types for ‘site’ and ‘component reference’

Conditional Expression

ConditionalExp	*Exp* `if` *Exp* `else` *Exp* *Exp* `unless` *Exp* `else` *Exp* `(if (`*Exp*`)` *Exp* `else` *Exp*`)`

This is your basic conditional expression. The syntax is borrowed from Python. It works just the same as the conditional statement, including bindings.

```
1 if p(x) else z+2
+x if L ~ [x, $(x)] else null
```

The precedence of the conditional expression is such that it occasionally requires parentheses around it.

Components

ComponentExp	*ComponentCtor*
	RPCExp
ComponentStmt	**`serve`** (**`before`** *ServeArgs? Stmt)* ¿ (**`after`** *ServeArgs? Stmt)* ¿ **`catch {`** *Case** **`}`** ¿ *Timeout* **`;`**
	SignalStmt
	SendStmt
	RecvStmt
ComponentCtor	**`spawn`** *[Id]* **`{`** *ComponentItem** **`}`**
	`spawn` *Id* **`(`** *Actuals* **`)`**
ComponentDecl	**`component`** *Id* **`(`** *Formals* **`) {`** *ComponentItem** **`}`**
ComponentItem	*ObjMember*
	(**`sync | async`** *) DistFun*
	`stable {` *FieldDef* **`* }`** --- **TODO :** **`stable`** modifier
	`initially` *Stmt*
	`reinit` *Stmt* --- ***TODO***
	`body` *Stmt*
	Import
AsyncStmt	*Exp* **`<--`** *Id Actuals*
SyncExp	*Exp* **`<->`** *Id Actuals [***`timeout (`*** Exp* **`)`***]*
SendStmt	*Exp* **`<<<`** *Exp*
RecvStmt	**`recv {`** *Cases Timeout* **`}`**
Timeout	**`timeout (`** *Exp* **`)`** *ExpOrBlock*
DistFun	*Id FunCases*
ServeArgs	**`(`** *Id [, Id]* **`)`**

`spawn` creates and starts a process.

The optional *Id* is a print name for the process, or, strictly, part of a print name for it. It can also be used on the closing tag of the **`spawn`**. It has no other significance.
`spawn CP(args)` spawns a component defined by a **`component CP {...}`**.
`component CP {...}` defines a component for later **`spawn(CP)`**ing.

<<< notes:

The receiver can be a site, a component, or a list thereof. If it's a list, it sends a message to each element of the list. This is a convenience, not an atomic multicast.
<<< calls the method `` `<<<` `` in the usual way.

`serve` notes:

The **`before`** and **`after`** blocks can have one or two formals. The first formal, if present, is bound to the envelope, inside that block. The second formal, if present, is bound to the sender.
If there is no **`catch{}`** clause, **`serve`** catches all errors, prints them, and pays them no further attention. If there is a **`catch{}`** clause, it is used as the **`catch{}`** of a **`try`**{}.

`sync` notes:

If you want to do a split sync, transform your code thus:

```
//BEFORE
//Caller
  x = server <-> sink();
// Server
  sync sink() {
    println("sploosh!");
    "Splooshed!";
  }


//AFTER
// Caller is unchanged. That's crucial.

// Server tosses message to worker and throws a particular
exception:
// Server
   sync sink() envelope e {
     worker <-- sink(e);
```

```
                 throw splitSync();
               }

            // Worker accepts an async message, and uses syncReply(e,
            whatever)
            // to send back response to e.
              async sink(e) {
                println("sploosh!");
                syncReply(e, "Splooshed!");
  }
```

TODO: write this!

```
// Compute the length of list L in a very stupid way indeed.
p = spawn{
  var n := 0;
  async bump(v) { n := n + 1;}
  sync tell()=n;
  var goOn := true;
  async plzDie() { goOn := false;}
  body{
    while (goOn) { serve; }
  }
  }spawn;

for (v <- L) p <-- bump(v);
n1 = p<-> tell() timeout(100);
p <-- plzDie();
```

Queries

Query	*QuerySwiss* *QueryQuantifier* *QueryFirstlike* *QueryTable* *QueryGroup* *QuerySort* *QueryGroup*
QueryControl	*QCFor* *QCI* *f* *QCVal* *QCVar* *QCWhile*
QCFor	**for** *Pat* **<-** *Exp*
	for *Pat* **<~** *Exp*
QCIf	**if** *Exp*
QCVal	**val**? *Pat* = *Exp*
QCVar	**var** *Id* **:=** *Exp* (**%<** | **%>**) *Exp*
QCWhile	(**while** |**until**) *Exp*
QueryControls	*QueryControl* ‡ **,**
QueryList	**%[** *Exp* . . .? **|** *QueryControls* **]**
QueryQuantifier	(**%count** | **%every** | **%some**| **%all** | **%exists**) **(** *Exp* **|** *QueryControls* **)**
QueryFirstlike	(**%first** **|** **%find**) **(** *Exp* [**%none** *Exp*] **|** *QueryControls* **)** (**first** **|** **find**) **(***QueryControls***)** *Stmt* [**else** *Stmt*]
QueryAfter	**%after** **(** *Exp* **|** *QueryControls* **)**
QueryTable	**%table(***QTKey*‡**,)** **{** *QueryTableItem** **|** *QueryControls* **}**
QueryGroup	**%group{** *QueryGroupItem** **|** *QueryControls* **}**
QuerySort	**%sort(***Exp QSortKeys* **|** *QueryControls* **)** **%sort[***Exp QSortKeys* **|** *QueryControls* **]**
QSortKey	**%>** *Exp --- descending order* **%<** *Exp --- ascending order*
QTKey	*Id* **=** *Expr*
QueryTableItem	*TableFieldWithInit*
QueryGroupItem	*TableId* **%count**

	TableId `%list` *Exp*
	TableId `%rev` *Exp*
	TableId `%first` *Exp* `%then` *Exp [*`%after` *Exp]*

TableId	`map` *?* `var` *Id* `:=`
	`map` *?* `val` *? Id* `=`

TableFieldWithInit	`map`*? (* `val`*?) (Id = Exp) ‡* `,` `;`
	`map`*? (* `var` *) (Id := Exp) ‡* `,` `;`

Synonyms:	`%< %none %then0`
	`%> %many %then1 %then`
	`%first %find`

All the queries have some *controls* (`for, if, val, var, while)`, and a *result clause*. The controls are the same for all queries. `for` controls iterate the way `for` statements behave. `if` controls skip iterations for which the test is false. `while` controls terminate iteration altogether (not just skipping a single one) when the test becomes false. `val` controls bind values in the usual way.

`var` controls vary a variable over the iteration. `var x := A %then0 B` initializes `x` to `A` before the first iteration, and on every iteration resets it to the value of `B`. For example,

```
  %after(s | for x <- L, var s := 0 %then0 s+x)
```

computes the sum of a possibly empty list of numbers. `var x := A %then B` is similar, but it initializes `x` on the first iteration, and only uses the value of `B` on the second and successive iterations. For example,

```
  %after(n | for x <- L, var n := x %then max(n,x))
```

computes the greatest element of a nonempty list. `%then` may be written `%then1` if you prefer.
If there are no iterations, `%then0` variables get their initial values; `%then1` variables get null, just like other uninitialized vars.
The `%after(`*E* `|` *Controls*`)` query runs through the whole loop, and then evaluates *E*. Only identifiers from `var` controls are available in *E*.

The list query `%[`*E* `|` *C*`]` produces a list of the values of *E*. The variant with an ellipsis, `%[`*E* `... |` *C*`]`, appends the results.

The quantifier queries `%count, %some`, `%every, %all, %exists` summarize how many times the result expression is true. `%count` returns the number of times; `%some` and its synonym `%exists` are true if the result expression is ever true; `%every` and its synonym `%all` are true if it is true on each iteration. `%some` and `%every` will quit early if the answer is known

```
%count(x>0 | for [x,y] <- L, if x mod y == 0);
%every(x > y | for x <- Xs, for y <- Ys);
%some(x.name == "Melyl" | for x <- people() );
```

The `%first` (which may be written `%find`) query returns the first values of the main result expression, or the value of the alternate expression if there were no values of the main one. For example, this searches for a person with the given SSN, and returns the first one found, or null if there are none.

```
%first(p %none null | for p <- people(), if p.ssn() == 123456789);
```

or more compactly

```
%first(p | for p && {:ssn:123456789:} <~ people())
```

(Note the use of `<~` here – it wouldn't work with `<-`)
The query returns null if there is no `%none` clause and no iteration found. Note that `%first` is simple and straightforward, and stops immediately if a first value is found.
`%first` may also be written as a statement, with keyword `first` or `find:`

```
find(for {:a,b:} <- tbl, if a.likes?(b)) {
   foundLike(a,b);
   }
else {
   nobodyLikesAnybody();
}
```

The `%table` query constructs a table. For example,

```
%table(a=i){ val sq = i*i; map var b := 0;
  | for i <- 1 .. 1000 }
```

constructs a table with, initially, a thousand elements. This table construction is identical to the obvious loop adding members to the table; in particular, if two key expressions have the same value, the second overrides the first.

The `%group` construct constructs a table as well, but by aggregating information. The key fields are the categories that the iterations are grouped into. The following example groups numbers from 2 to 12 into two categories, with `prime=true` and `prime=false`.

```
p = %group(prime = n.prime?()) {
          numbers = %list n;
          srebmun = %rev n;
          count = %count;
          |
          for n <- 2 .. 12
       };
```

The non-key fields hold aggregate information. There are a few choices. **`%count`** simply tells how many values fit that category. **`%list`** *E* fills that field with a list of the values of expression *E*; in the case of **`p(true).numbers`**, the list of primes. **`%rev`** *E* is the same list backwards; computing it is more efficient, and it is recommended in the cases where order doesn't matter.

The most general form is *Id* **`=%first`** *Expf* **`%then`** *Expt [***`%after`*** Expa]*. This works much like a **`var`** query control. On the first iteration that falls into the given group, *Id* is set to the value of *Expf*. On later iterations, it is set to *Expt*. After the last iteration, it is set to *Expa,* or, if that clause is absent, it is left unchanged. The following computes the list and the sum of squares of primes and nonprimes:

```
p2 = %group(prime = i.prime?()) {
             map numbers = %first [i]
                           %then [i, numbers...]
                           %after numbers.reversed();
                 sumsq = %first i*i %then sumsq + i*i;
             | for i <- 2 .. 11
             };
```

`%sort` produces a list of values of one expression, sorted by the values of others.
`%sort( x %< x.a() | for x <- L);` sorts the elements of **`L`** in order of increasing **`a()`** values.
`%sort( x %< x.a() %> x.b() %> x.c() | for x <- L);` sorts them by increasing **`a()`**, then decreasing **`b()`**, and then decreasing **`c()`**. The bracketed form is the same as the parenthesized form, and probably better to use, since it emphasizes that **`%sort`** produces a list.

Built-In Functions And Methods

Convention

- Predicates generally end in **`?`**.
- Mutation commands either have names that are unambiguously verbal (**`add`**) or end in **`!`** (**`add!`**). In the case of verbal names, we frequently allow both **`!`**ed and **`!`**less forms, as exact synonyms.
- Commands that return null when they fail, or a non-null value to indicate success, have no distinguished syntactic form; that is the standard form.

Everything

`x.str()`	String representation of **`x`**. This method is called whenever **`x`** needs to be converted to a string (*e.g.,* **`"$x"`**). Your classes should implement **`str`** if you want them to be printable.
`x.num()`	The best numeric representative of **`x`**, *e.g.,* the length of a list, the codepoint of a character, or the size of a table. Not all values have a best numeric representative.
`x.len()`	Same as **`x.num()`**, but intended for cases when the number is actually a length.
`x.invoke(M, args)`	Invoke the method named **`M`** with arguments **`args`** on **`x`**.

Boolean

`str()`	String representation of the entity.

Class

`str()`	String representation of the entity.

Closure

`str()`	String representation of the entity.

Float

`f.str()`	String representation of **`f`**.

`f.num()`	`f` itself
`f.within?(x,tol)`	|`f-x`| ≤ `tol`
`f.round()`	The nearest integer
`f.ceil()`	The nearest integer ≥ `f`
`f.floor()`	The nearest integer ≤ `f`

Integer

We could add the other binary manipulation features of java.lang.Integer, like rotate and count bits.

`n.num()`	Returns the number `n` itself.
`n.prime?()`	A primality test. Not particularly efficient.
`n.str()`	String representation of `n`.
`n.bitAnd(m)`	Bitwise and with `m.`
`n.bitOr(m)`	Bitwise or with `m.`
`n.bitXor(m)`	Bitwise xor with `m.`
`n.bitNot()`	bitwise negation
`n.bitShl(m)`	shift left by `m`
`n.bitShr0(m)`	shift right unsigned (>>> in Java). Shifts 0 in from the right.
`n.bitShr(m)`	shift right signed (>>)
`n.bits(i,j)`	Get bit field from position `i` to `j`.
`n.bitSet(i,j,v)`	Return a new integer which is `n` with the bit field from `i` to `j` set to `v`
`n.strHex()`	The hexadecimal representation of `n` (with no leading `0x`)
`n.strOct()`	The octal representation of `n` (with no leading `0o`)
`n.strBin()`	The binary representation of `n` (with no leading `0b`)
`n.strBase(m)`	The base-`m` representation of `n`
`n.sgn()`	Signum of `n`: -1 for negatives, +1 for positives, and 0 for 0.
`n.abs()`	Absolute value of `n`
`n.pow(m)`	Power n^m. Requires `m` ≥ 0.
`n.max(m)`	Maximum. `m` may be a list.
`n.min(m)`	Minimum. `m` may be a list
`n.trim(i,j)`	`n` if between `i` and `j`. `i` if below; `j` if above.
`n.rand0()`	Return a random number between `0` and `n-1`, inclusive
`n.rand1()`	Return a random number between `1` and `n`, inclusive
`n.round()`	
`n.ceil()`	
`n.floor()`	
`n.char`	Return the `char` with `n` as codepoint.

List

`L.len()`	Length of the list `L`
`L.num()`	Length of the list `L`
`L(i)`	Get the `i`th element of `L.` If `i` is negative, count from the right; so `L(-1)` is the last element of `L`.
`L(i..j)`	Same as `L.sub(i,j)` but only if 0 ≤ i ≤ j
`x :: L`	A new list whose head is `x` and whose tail is `L.` Same as `[x, L...]`
`L1 @ L2`	A list obtained by appending `L1` to `L2`. Same as `[L1..., L2...]`
`L.range()`	The range of subscripts of the list `L`
`L.reversed()`	The list, reversed. `[1,2,3].reversed() == [3,2,1]`
`L.reverse()`	A synonym for `reversed()`
`L.str()`	String representation of `L`.
`L.rand()`	Return a random element of `L`.
`L.unique()`	Return a list with the elements of `L`, without duplicates. Order not guaranteed.
`L.setEq(L2)` `L.setEq?(L2)`	True if `L` and `L2` have the same elements.
`L.subset(L2)` `L.subset?(L2)`	Every element of `L` is an element of `L2`.
`L.union(L2)`	A set-theoretic union of `L` and `L2`: *viz.* a list whose elements are those of `L` and `L2`, in arbitrary order, without duplicates.
`L.intersect(L2)`	A list whose elements are those in both `L` and `L2`. Order not guaranteed.

`L.minus(L2)`	The elements of `L` which are not in `L2`; order preserved; duplicates allowed.
`L.zip(L2)`	Return a list whose elements are pairs `[L(i), L2(i)]`.
`L.index(x)`	Return null if x isn't an element of `L`, or the position of the first occurrence if it is.
`L.followedBy(x)`	`[L..., x]`
`L.joined(s)`	`== s.join(L).` That is, return a string obtained by `str()`inging all the elements of `L`, and concatenating those, with a copy of `sep` between each pair.
`L.and()`	Returns the elements of `L`, stringified and separated by commas and the word "and" to make a proper English list according to Strunk and White's *The Elements of Style*.
`L.tails(N)`	Returns the list of tails of `L` whose length is at least `N`. `[0,1,2].tails(0) = [[0,1,2], [1,2], [2], []].` `[0,1,2].tails(2) = [[0,1,2], [1,2]].` Useful for looping over distinct elements and such.
`L.tails()`	`L.tails(0)`
`L.tail()`	The second and following elements of `L`. Null for the empty list.
`L.head()`	The first element of `L`. Error for empty list.
`L.flat1()`	Flatten one level of list structure: `[1, [2,3], [4, [5], []]].flat1() == [1,2,3,4,[5],[]]`
`L.flat()`	Flatten all levels of list structure. `1, [2,3], [4, [5], []]].flat() == [1,2,3,4, 5]`
`L.interleave(L2)`	Produce a list of alternating elements of `L` and `L2`. `L2` must be either the same length as `L`, or one element shorter.
`L.numbered()`	Produces a list of tuples `[{: k:0, v:L(0) :}, {: k:1, v:L(1), ...].` This is useful for iterating over a list and positions of elements in it.
`L.asBytes()`	Turn a list of small integers into a `bytes` value.
`L.sub(i,j)`	Return elements number `i` through `j`, inclusive.
`L.subx(i,j)`	Return elements number `i` up to but not including `j`.
`L.sublen(i,n)`	Sublist of length ≤ `n` starting at element `i`.
`L.left(n)`	The first n elements, or all of `L` if there aren't that many elements.
`L.butleft(n)`	Everything but the first n elements
`L.right(n)`	The last n elements
`L.butright(n)`	Everything but the last n elements.
`L.sum`	The sum of the elements of `L`. `[].sum == 0; [x].sum == x`. Aside from that, `L.sum` is like putting `+` between each pair of adjacent elements of `L`. So, `[1,2,3].sum==6; [1,"ow"].sum == "1ow".` The result is `null` if any partial sum is `null` (which will not happen with a list of integers, but might happen with user-defined objects).
`L.cat`	The concatenation of the elements of `L`, converted to strings. Same as `L.joined("")`.

Module

`str()`	String representation of the entity.

Table/Map

Most table operations return null if there's no important value to return.

`clear!`	Remove all rows from the table.
`ins(r)`, `ins!(r)`	Insert or update a row. `r` must have at least the fields of the table (keys and non-keys); other fields are ignored. **TODO:** Also works on a list of rows.

`del(r)`, `del!(r)`	Delete a row. `r` must have at least the key fields of the table. Other fields – including ones that the table also uses – are ignored.
`get(r)`	Find a row of the table, using the key fields of `r` as a key. Returns null if there is no such row.
`insNew(r)`, `insNew!(r)`	Insert a row. It is an error if there is a row with the same key already in the table. Also works on a list of rows.
`delOld(r)`, `delOld!(r)`	Delete a row. It is an error if there is no row with that key. Also works on a list of rows.
`insExact(r)`	Insert a row into a table. The record `r` must have the shape of a row; that is, it must be a record whose fields are precisely the keys and non-keys of the table. **TODO: not implemented.**
`delExact(r)`	Delete a row. `r` must be a record with the shape of a row. In the latter case, that record must exactly match one already in the table. **TODO: not implemented.**
`keys()`	A list of records: the keys of the rows currently in the table.
`rows()`	A list of records: the rows currently in the table.
`lst()`	A list of the `map` elements. TODO: really?
`anyRow()`	Returns a row of the table, or `null` if the table is empty. There are no guarantees about repeated calls: you may get the same or a different row on the second call. This is potentially more efficient than calling `rows()` and then picking a row.
`insAll(rs)`, `insAll!(rs)`	Insert all the rows in a list.
`delAll(rs)`, `delAll!(rs)`	**TO DO**
`toRecord()`	Only works for a table with one key and one non-key field. Returns a record whose field names are the `str()`s of the key and whose field values are the non-key field. It is an error if two keys have the same `str()`.
`str()`	String representation of the table.
`same?(t)`	True if the two tables are the same table.
`== t`	True iff the two tables have the same set of column names, and the sets of rows are equal.
`has?(x)`	True if `x` is a key of `tbl`. It is an error to call this on a table with more than one key field.
`num`, `len`	The number of elements in the table.

Ords

Most ord operations return null if there's no crucial value to return.

`add(r)`, `add!(r)`	Add record `r`, which must have at least the fields of the ord. (Only the relevant fields are added, of course).
`addAll(L), addAll!(L)`	Add a list of records.
`add1(v), add1!(v)`	Another syntax for `@=`
`add1All(L), add1All!(v)`	`@=`'s all the elements of L
`ot @= v`	Adds a record with map field equal to `v` and all other fields null, to a table which has a map field. Returns null.
`rows()`	Returns the rows of the table, as a list. (In order, of course.)

`o[k]`	Get the map field of element number `k`. Fails if `k` is not in bounds.
`o(k)`	Get the whole row numbered `k`. Returns null if `k` is a number but not in bounds.
`o[k] := v`	Mutate row `k` so that its map field is `v`. Error if that field is not mutable, or if it does not exist, or if map access is forbidden.
`o(k) := r`	Mutate the ord so that extant row `k` is record `r`. `r` must have all the fields of the table; other fields are ignored. If `k` is the index of a row of the table, that row will be replaced. If `k` is negative, it counts from the end. Unlike for tables, this operation does *not* work if `k` is not an index into the ord.
`o(k) := null`	Delete the row currently numbered `k`. If you delete a row in the middle of the ord, a new row will thereafter be numbered `k`.
`ot.addAfter(k, r)`	Mutate the ord to insert a new row after position `k`. Adding after the end of the list adds to the end of the list.
`ot.addBefore(k,r)`	Mutate the ord to insert a new row before position `k`. Adding before index 0 always inserts at the beginning of the list, even in an empty list. Adding before a position past the high end of the list adds at the end of the list.
`ot.del(k)`, `ot.del!(k)`	Delete row number `k`. If `k` is a range, deletes rows in that range. If `k` is a number, returns the deleted row.
`ot.pos(k)`	Return the positive (strictly, non-negative) form of index `k`; that is, the number ≥ 0 such that `ot(ot.pos(k))` gets the same record as `ot(k)`. It is an error if `k` is out of bounds. This is particularly useful with ranges: `ot.del(3 .. ot.pos(-1))` deletes all but the first three elements of `ot`.
`forbidMapSet(b)`	As for a table: if `b` is true, `o[k]:=v` and `o @= v` will throw an error. This is intended for supporting upgrading an ord from one to many fields.
`ot.lst()`	Return a list of the `map` field values.
`ot.clear!()`	Remove all rows.
`ot.range()`	Range of subscripts for the ord **TODO!**.
`ot.remove(x)`	Removes the first entry whose map field is equal to x in ot.
`same?(ot)`	Returns true if this and ot are the same ord.
`len`	current length of the ord

Range

Ranges have the structure *min* `..` *max*, where *min* ≤ *max,* or the special empty range 0..-1. Any attempt to make a range with *min*>*max* will result in the special empty range.

`num()`	Returns max-min+1, *viz*. the number of elements in the range.
`str()`	String representation of the entity.
`empty?()`	Is this range the empty range?
`trim(m)`	Returns m if m is inside the range; min if m is below the range, and max if m is above the range.
`r.min`	Lowest value in the range
`r.max`	Highest value in the range
`r.rand`	Random number in the range

Record

`str()`	String representation of the entity.
`toMap()`	Returns a map whose keys are the field names of the record and whose values are the values.

Closure

To do: We need arity and apply.

String

Subscripting a string with a non-negative integer gives the character at that position. Negatives count from the end, so `s(-1)` is the last character of `s`. Two-argument subscripting with nonnegatives is just like Java's substring operation: `"0123"(0,2) = "012".` Negatives count from the end here too, but with negatives the endpoint *is* included, so `s(1,-1)` returns all but the first character of `s`, and `s(1,-2)` returns s except the first and last characters. Numbers off the end of the string are counted as being at the end of the string.

I'm intentionally trying to stay close to the Java interface.

`<=>(s)`	Same as java.lang.String.compareTo()
`compareToIgnoreCase(s)`	Same as Java.
`contains?(s)`	Is `s` a substring of this. Note different name than Java.
`startsWith?(s)`	Java's.
`endsWith?(s)`	Java's
`codePointAt(n)`	Java's.
`equalsIgnoreCase?(s)`	Java's. Note good Thorn style of predicate.
`index(s)`	The position in `this` of the start of the first occurrance of `s` as a substring, or null if there is none.
`indexAfter(s,i)`	The position in `this` of the start of the first occurrance of `s` after (or possibly starting at) position `i` as a substring, or null if there is none.
`num()`	Length of the string
`replace1(s,r)`	Returns a new string identical to this except that the first occurrance of substring `s` is replaced with `r`. **TODO: this isn't as convenient as the others in Java, so I didn't do it yet.**
`replace1RE(re, r)`	Returns a new string identical to this except that the first match of regexp `re` is replaced with `r`.
`replace(s,r)`	Returns a new string identical to this except that all occurrances of substring `s` is replaced with `r`. If `s` is empty, this inserts a copy of `r` between every pair of characters in `this`.
`replaceRE(re,r)`	Returns a new string identical to `this` except that all matches of `re` are replaced with `r`.
`s(n)`	Return the n'th character of s. Negatives apply here. If `n` is a range `i..j`, this returns the substring from character `i` to character number `j`, just like `s.thru(i,j)`. But, because this is a range, it only works if 0 ≤ `i ≤ j`.
`sub(i,j)`	Return characters number `i` through `j`, inclusive.
`subx(i,j)`	Return characters number `i` up to but not including `j`.
`sublen(i,n)`	Substring of length ≤ `n` starting at character `i`.
`left(n)`	The first n characters, or all of the string if there aren't that many characters.
`butleft(n)`	Everything but the first n characters
`right(n)`	The last n characters
`butright(n)`	Everything but the last n characters.
`split(re)`	Java's.
`startsWith?(s)`	Java's.

str()	String representation of the entity.
trim()	Java’s.
decodeInt()	cf. Java’s Integer.decode(string)
int()	Parse the string as an integer; return null if it doesn’t parse.
s.float	Parse the string as a float (which may give an integral value; ‘1’.float == 1.0). Return null if it doesn’t parse.
intBin()	Parse the string as an integer in the appropriate base (binary, octal, hexadecimal, or n); return null if it doesn’t parse.
intOct()	
intHex()	
intBase(n)	
matchRE(re)	Returns null if this fails to match regexp re. If the match succeeds, it returns a list of records {: start, end, text :}. The first element of the list is the start, end, and text of the whole string matched; the remaining ones correspond to captured groups.
matchRE?(re)	Returns true/false if this matches/fails to match regexp re.
matchREAfter(re,p)	Like matchRE, but starting at position p.
matchREAfter?(re,p)	Like matchRE?, but starting at position p.
toUpper()	
toLower()	
capitalize()	
`match/`(re)	Returns null if this fails to match regexp re. If it does match, it returns a list of strings corresponding to the groups that appear in re – *not* including group 0.
plusChar(c)	s.plusChar(nc) returns the string s with the character whose codepoint is nc appended to it: “”.char(97)==”a”.
char(i)	Get the ith character from this string, as a char value. This returns a true Unicode-code-point kind of character. If you really want to do low-level Java-style character manipulations, use javaChar.
s.withSub(i,j,t)	Answers a string just like s except that where s.sub(i,j) is in s, it has t.
s.withSubx(i,j,t)	A string like s but with s.subx(i,j) replaced by t
s.withSublen(i,n,t)	A string like s but with s.sublen(i,n) replaced by t
s.range	The range of subscripts allowed for s, *viz.* 0..(s.len-1)
s.javaChar(i)	The java family of character operations treats a string as an array of 16-bit java-chars (here represented as integers). Most of them are codepoints, but some Thorn/Unicode characters are encoded as a *surrogate pair*. If you don’t know about surrogate pairs, read about them before using these functions. s.javaChar(i) gets the i’th javaChar from the Java-style representation of s.
s.javaLen	The number of javaChars in s.
s.javaPlus(L)	Returns a string which is s with the integers in L (treated as javaChars) concatenated onto the end of it.
s.format(x,y,...,z)	Formats the values x,y,...z, using s as a format string. This essentially calls Java’s String.format function, converting some Thorn built-in types to the corresponding Java types. format can take any number of arguments.
s * n	Concatenates n copies of s
s.quote	A version of s that can be read by Thorn as being the same as s.
s.deJSON	Parses a JSON-encoded value into a Thorn value.

.

Site

name()	Internet node name
port()	Port

`sendln(s)`	Send the string `s`, with a newline added if necessary, to the site. This uses lower-level net protocol, not the usual Thorn to Thorn communication.

File

`str()`	

`num()`
`len()`
	The length of the file, in bytes.

`abs()`	
`path()`	
`writeln()`	
`del!()`	
`flush()`	
`exists?()`	
`contents()`	The whole contents of the file, perhaps with a newline appended.
`clear!()`	Erase the contents of the file.
`dir?()`	
`parent()`	
`asDir()`	
`rename()`	
`hashCode()`	
`writeBytes(bytes)`	Write the bytes in a low-level way to this file.
`readBytes()`	Read a file as a bytes object
`setContents(s)`	Make the whole file be `s.`
`name`	Last component of file path; cf Java getName()
`mkdir`	Create a directory.

Dir

All the `file` methods work on dirs too (perhaps unwisely). In addition:

`files`	A list of all the files in `this`.
`file(s)`	Get a `file` pointing at a file named `s` in `this`: it need not exist.

`write,`
`writeBytes,`
`setContents`
	(Doesn't work – you can't write the contents of a directory)

`mkdir`	Create a directory.

Char

Char values (which are pretty much java.lang.Character) provide Unicode information. They correspond to Unicode Scalar Values: integers in the range 0 to 10FFFF (hex), corresponding to java.lang.Character.isValidCodePoint(). A single char may need to be expressed as two `java.lang.Character`s, or four bytes in a string, but this is transparent to Thorn programmers unless they want to look.

`str()`	Turn the character into a one-element string
`num()`	Get the numeric value of the Unicode code point
`< c2`	chars are ordered
`category`	Get the category for the character, as a short string: *e.g.*, `"Ll"` for lowercase letters, `"Ps"` for start of punctuation, etc. Corresponds to java.lang.Character.getType().
`dirType()`	Returns the Unicode spec's bidirectionality character type: "AN" for Arabic Number, etc. **TODO: If we were kind we would have an includable library around this to interpret it.**

`directionality`	Returns the short string directionality character type for this character. E.g., neutral bidirectional characters have `directionality="B"`. **Warning:** This relies on java.lang.Character.getDirectionality, which sometimes gives different answers from http://unicode.org/Public/UNIDATA/UnicodeData.txt (For example, the plus character is listed as having directionality ES, but Java and Thorn think it is ET.)
`defined?`	True if the character is defined in Unicode.... Note that you can't even create the character from an integer codepoint if the number isn't of the right size (up to 10FFFF). But not all codepoints in that range correspond to actual characters in Unicode, hence this function.
`digit?`	sTrue if this is a digit.
`identifierIgnorable?`	Should this char be ignored in Java/Unicode identifiers?
`control?`	
`letter?`	
`letterOrDigit?`	
`lowercase?` `lower?`	
`mirrored?`	
`whitespace?`	
`title?`	
`unicodeIdentifierPart?`	
`unicodeIdentifierStart?`	
`upperCase?` `upper?`	
`lower` `lowerCase`	Lowercase version of this
`upper` `upperCase`	
`title` `titleCase`	
`block`	The name of the Unicode block containing this character.
`+`	concatenation returns strings in a sensible way
`asSurrogatePair`	**TODO.** If this character needs to be represented as a surrogate pair, return a two-element list with the integers corresponding to the halves of surrogate pair. Otherwise, null

Bytes –

`bytes` is a minor type used to wrap an array of bytes. It is used for low-level operations. **TODO:** I've left it uncertain whether it is mutable or not. Currently they are neither immutable (officially) nor do they have any mutation operations.

`str()`	Turn the bytes into a String.
`num()`	How many bytes are there?
`(i)`	Get byte number i, counting from 0, as an integer
`len()`	How many bytes are there?
`lst()`	Turn a bytes into a list (of small integers)

Built-In Functions

These aren't methods on anything in particular, so they're phrased as functions.

`thisSite()`	The current site.
`thisComp()`	The current component.
`site(n,p)`	Create a site from a string (node name) and number (port)
`site(uri)`	Create a site from a URI. **Current:** just gacks the host and port out of the URI and ignores the rest.
`syncReply(enve, v)`	If `enve` was the envelope of a `<->`, this sends back a response of `v` to it.
`splitSync()`	Returns the value of exception recognized by the recv command for doing a split sync.

`crude_serialize(x)`	Serialize x the Java way.
`crude_deserialize(x)`	Deserialize a `bytes` the Java way.
`parseConstant(s)`	Reads a Thorn constant from `s`. *E.g.*, `parseConstant("[1,true]") = [1,true]`
`same_type_and_all_fields_equal?(a,b)`	True of two records or objects if they have the same field names or class, respectively, and all their fields are `==`. This is useful for defining `==` inside a class. It can violate abstraction barriers – it doesn't use the nullary method calls, but the actual fields. As with many functions with very long names, you're not really supposed to call it much yourself.

Extensions: Defining Functions and Classses from Java

JavaFunDecl	`javaly fun` *Id* `(` *Ids* `)` `=` *QualName* `;`
JavaClsDecl	`javaly class` *Id [*`(` *Ids* `)`*]* `=` *QualName* `{` *(JavalyMethodDecl | JavalyNewDecl) ** `}`
JavalyMethodDecl	`def` *Id* `(` *Ids* `)` `=` *Id* `;`
JavalyNewDecl	`new` *Id* `(` *Ids* `)` `;`

A javaly function declaration connects a Thorn identifier to a Java public static method. The formals of the javaly function (in Thorn) must be identifiers; their names have no significance beyond documentation and arity. The QualName indicates a package, class, and function, following the usual Java rules. So,
`javaly fun println(x) = fisher.runtime.builtInFun.IO.println;`
indicates the package `fisher.runtime.builtInFun`, the class `IO`, and the public static function `println(Thing)`. In Java, all the arguments and the return value of the function must be `Thing`.
The formals in a *JavaClsDecl* serve as both the visible fields of the object, and the subpatterns in the standard extractor pattern.
TODO: Explain that javaly classes have to extend ThingExtended (if that is true).
Certain methods are automatically attached to Java methods defined by ThingExtended:

`str()`	Attached to `str()`, which calls Java's `toString()`
`==()`	Attached to `eq()`, which calls Java's `equals()`
`hashCode()`	Attached to `th_hashcode()`, which calls Java's `hashCode()`.

XML

XML support is defined in the XML module in the standard library.

`parseXML(src)`	Parses `src` into XML objects. `src` may be a string, which will be parsed directly; or it may be a File, whose contents will be parsed
`Elem`	The `Elem` class describes XML Elements. It has fields: `tag`: a string giving the element's tag `attrs`: a record giving the element's attributes `children`: a list giving the element's children The constructor `Elem(t, a, c)` makes new elements, and serves as a pattern to deconstruct them.
`no Text`	TEXT nodes are converted into strings internally.

The `str()` function on `Elem`s can be given a record of values to provide extra control over printing, as specified in http://www.w3.org/TR/xslt#output. The field names and values of the record must be the same as the XSLT processor attributes – which means that, to omit the XML declaration, you'd use `x.str( {: `omit-xml-declaration`: "yes" :} )`. Values are converted to strings with `str()` first.

Elems have the following:

`e.kid(kidTag)`	**TODO-tests** Return the unique child whose tag is kidTag, or null if there is no such unique child.
`e.kids(kidTag)`	**TODO-tests** Return the list of children whose tag is kidTag

Line-Based Non-Thorn Communication (Experimental,

incomplete)

First, you need the following:
```
module LINES {
  javaly class StringSocketeer =
fisher.runtime.lib.socketeer.StringSocketeer {
    new StringSocketeer(port);
  }
  javaly class ReplySocket = fisher.runtime.lib.socketeer.ReplySocketTh
{
    def `<<<`(x) = send;
  }

}
```

Then, you can open a listener for line-based communication on a given port with:
```
ss = StringSocketeer(4321);
```

And, after that, lines of input coming in on that port show up as string messages in your mailbox. They are `from` a `ReplySocket`. You can send messages to the `ReplySocket` with the low-level `<<<` operation. Here's a noisy not-quite-echo sort of program:
```
ss = StringSocketeer(4321);
  var n := 0;
  while(true){
     recv{
       x from y => {
          println("x=$x");
          println("y=$y");
          y <<< "Owie! Why do you say $x?\n";
          println("I owied.");
       }
     }
  }
```

Known Bugs

The immutability detector is oversensitive. Objects, modules, and closures are assumed mutable in many situations that they are actually immutable.
The order of evaluation of field initializers and constructors in classes is messed up. Currently, all initializers are evaluated first, then the constructor. This is sometimes right and sometimes wrong. The right order is based on dependencies, which I am not currently doing. **Symptom:** failed attempt to get the value of a formal parameter (usually) out of a frame. **Workaround:** Explicit constructor with explicit order of evaluation.

File Encodings

Thorn currently uses UTF-8 for everything: Thorn source files, and files it reads. **TODO:** Make this parameterized in useful ways. Thorn source should get a command-line option. Methods like string's `file()` should get an optional argument (or alternate method call)

JSON notes

```
module JSON {

  javaly fun jsonParse(s) = fisher.runtime.lib.json.JSON.parse;

  javaly fun json?(x) = fisher.runtime.lib.json.JSON.isJSON;

  javaly fun asJson1(x) = fisher.runtime.lib.json.JSON.asJSON;

  javaly fun asJson(x,conv) = fisher.runtime.lib.json.JSON.asJSON;

  javaly fun jsonRecordize(x) =
fisher.runtime.lib.json.JSON.jsonRecordize;

  javaly fun jsonRecordizeWithClassName(x) =
fisher.runtime.lib.json.JSON.jsonRecordizeWithClassName;



}
```

- `jsonParse(s)` returns the parsed version of `s` as a JSON object in Thorn – using records and lists. It returns `null` if `s` doesn't parse.

- The string method s.deJSON does the same thing.

- **`json?(x)`** returns true if **`x`** is a JSON-compatible object.

- **`asJson1(x)`** returns **`x`** if **`x`** is JSON, and throws an error if not; probably not very useful

- **`asJson(x, conv)`** attempts to convert **`x`** to a JSON object. **`conv`** is either a conversion function, or null. If it is null, **`asJson`** uses a standard translation **WHICH SHOULD BE DESCRIBED.** If it is supplied, it will be applied to non-JSON data; its result should be a JSON value.

- **`jsonRecordize(x)`** is a good choice for conversion function; it transforms an object **`x`** into a record whose fields are the fields of **`x`**, run through **`asJson(y, jsonRecordize).`** Somewhat incidentally, jsonRecordize also does asJson to non-object arguments. **Warning:** This does not currently handle objects with cyclic references.

- **`jsonRecordizeWithClassName(x)`** is another good choice for conversion function. It produces records as **`jsonRecordize(x)`** does, but they are augmented with the name of the class of **`x`** when that is available, in a field named **`` `class` ``**.

Pure Classes and Transmission of Data

An object may be *sent* if it is pure.
An object may be *received* if:

Its class is loaded into the receiving component at top level; *i.e.*, imported directly or a member of an imported module.
Its class has the same hash code as the class being sent.

Networked Communication with NetSite and ListenOn

Warning: This is work in progress. The implementation is particularly crappy, and very badly tested. Corrections and improvements are welcome.

Sending and receiving arbitrary messages over the network is done by the NetSite and ListenOn classes. Both classes use a Protocol object which describes the network protocol: the **`encode(s)`** method is applied to outgoing messages, and the **`decode(s)`** method applied to incoming ones.

Other fields of the protocol object control certain fine points of sending and receiving. Currently we allow (a) HTTP, and (b) single-line messages.

The classes are defined in fisher/lib/universal/NET.thm

The situation is not symmetrical.

- If you're *sending* a message, it will go to some particular IP address and port; you will first construct a NetSite for that address and port, and use the NetSite to send messages.
- If you're *receiving* a message, you will be listening on a particular port. You construct a ListenOn object, which directs incoming messages on that port into your mailbox.

Here's an example of sending single lines on port 4444. The protocol here prepends the string Yow! on each line sent. (Which is silly, but you might well, e.g., put some useful delimiters there, or convert newlines to non-newlines to allow transmission of multi-line messages, or whatever)

```
import NET.*;
  protox = object extends Protocol {
    def encode(msg) = "Yow! $msg\n";
    def decode(msg) = (throw "Not used";);
    def input() = UseLINE;
  };
  sx = NetSite("localhost", 4444 , protox);
  sx <<< 'Some message';
```

These could be read by:

```
  import NET.*;
  protoy = object extends Protocol {
     def encode(msg) = (throw "Not used";);
     def decode("Yow! (.*)"/[orig]) = orig;
     def input() = UseLINE;
  };
  sock = ListenOn(PORT,protoy);
```

Note that the **`decode`** strips off the **`Yow!`** that the other **`encode`** added.

Using HTTP is similar: The following does a simple `GET /whatever?q=splash` with no extra headers or content. HTTP messages get content-length supplied automatically; all other headers are manual. (Currently).

```
import NET.*;
  protoclient = object extends Protocol {
    def encode(msg) = msg;
    def decode(msg) = msg;
    def input() = UseHTTP;
  };
  sirvir = NetSite("localhost", PORT, protoclient);
  req = HTTPRequest("GET", "/whatever", {:q: "splash":},
            {: :}, "");
  sirvir <<< req;
```

To send an HTTP response:

```
 r1 = HTTPResponse(404, "File Not Flounder", {::}, "Clever message");
 sirvir <<< r1;
```

If you want to use JSON... currently you have to manage conversion to and from JSON yourself. The method `str.deJSON()` un-JSONs a string into an object. There currently isn't any convenient way to turn values into JSON, but that's the easy direction.
Note that you can use JSON in a header (if there aren't newlines) or in the content of a HTTP message.