Jumbo ML

Smooth Sailing to Module Mastery

Norman Ramsey, Tufts University

On July 31, 2014, I talked about Jumbo ML with a very vigorous audience of researchers and teachers from Harvard and Northeastern. Many interesting things are said, and my notes are both distributed and collected at the end, in italics.

Problem, Part I: Teaching programming

A computer scientist should be able to prove theorems and write programs. Most introductory instruction focuses on programming. A great strength of this instruction is that students actually build programs. But building requires materials: a technology for teaching programming. And too often, we ask students to use the same technology that industrial engineers use. Unfortunately, when beginning students use an industrial-strength programming language, the difficulties of mastering the language divert students from intended learning outcomes. Beginning students should be provided with a "teaching language" tailored to their needs. Using a suitable teaching language, the essential principles that are taught in an introductory course should be clearly and easily made manifest.

At present there is a thriving ecosystem of languages designed for teaching absolute beginners, usually in middle school or high school. One well-known example is Scratch. At the university level, I am aware only of *How to Design Programs* and its three teaching languages Beginning Student Language, Intermediate Student Language, and Advanced Student Language. These languages ship with tools, a textbook, and a design method, and the result is very effective. They are a great way to get students started with deep ideas about programming. But they can only carry you so far: they are missing much of what we'd like to teach in the second course.

Problem, Part II: The second course

To talk about the first and second courses in computing, ACM curricula use the words "CS1" and "CS2." Instructors generally agree that CS2 means some sort of course in basic data structures, but they may differ on the details. I, too, view CS2 as a data-structure course, but I don't view data structures as foundational. I believe that data structures follow from two more fundamental concerns: abstractions and cost models. And the most critical abstraction is the module abstraction. The fundamental ideas are laid out nicely in Butler Lampson's *Hints for Computer System Design*: programs are composed of *interfaces* and *implementations*, interfaces define *abstractions*, and *client* code uses the abstraction. A good abstraction provides not just a clean interface but also a perspicuous and helpful *cost model*.

What do all these ideas have to do with data structures? A data structure *follows* from a choice of abstraction and a cost model. For example, if you want a bag abstraction with fast access to the smallest element, you want a heap. Similarly, if you want an ordered-list abstraction with fast insertion anywhere but with removal only of the first element, you also want a heap! My slogan is

Abstraction + Cost Model = Data Structure

With this organization in mind, here are my goals for CS2:

- Students will build programs from modules, will understand how modules are connected, and will have an idea how to solve large problems by connecting modules.
- Students will be able to estimate how much time and space a program needs for its execution. Moreover, students will be able to manage time and space costs by shifting them to the most appropriate part of a system.
- Students will become comfortable with some *data structures* that are widely used in many modules. These data structures are a common currency of late 20th-century industrial computing culture, and they are *very* popular with phone screeners and job interviewers, as well as programmers.

The role of Jumbo ML is to support these learning goals while remaining as simple and as easy to learn as possible.

Language needs for CS2

What kind of language we want depends on what we want students to learn. The key learning goals that affect my own language choices are programming with abstraction, reasoning about costs, and understanding the decomposition of programs into modules.

Students will be able to build substantial programs only if they can use abstraction. The most fundamental abstraction is *procedural* abstraction: students need to be able to call a procedure (function, subroutine) knowing only its specification, not its implementation. I call this specification a *contract*; other writers use *purpose statement* or *precondition and postcondition*. If contracts are important, then we should lean toward pure functional languages: contracts for pure code are *much* simpler than contracts for impure code. (Try writing the specification for a mutable stack. Then try an immutable stack.)

If we want students to be able to reason about costs, then the programming language needs a perspicuous cost model. If this were the only criterion, I would teach the second course in C, which enjoys a cost model of unparalleled perspicuity. Among popular functional languages, Scheme probably has the simplest cost model. Haskell's cost model makes grown persons weep.

Finally, if students need to understand how programs are decomposed into modules, they should be able to look at interfaces. And I want interfaces to be separately compiled. A person could limp along with C's `.h` files, but I want to rule out the lines taken by C++, Clu, Haskell, Oberon, Racket, and a bunch of others, where all the compiler understands is an

implementation, and certain parts of the implementation are called *public*, *exported*, or *provided*. If we want students to learn information hiding, they need to look at code that hides information.

My colleagues and I are not aware of any available language that meets all our criteria. Among the languages that are available, the best choice seems to be the dead research language Standard ML:

- It militates toward pure code but also supports impure code and mutable abstractions.
- I wouldn't call the cost model perspicuous, but at least it's discoverable.
- It provides first-class, separately compiled interfaces.
- While significantly simpler than its living relatives Haskell and OCaml, it is nowhere near as simple as real teaching languages. Fortunately, having taught using Standard ML, I know some of the pitfalls.

What simplicity looks like: *SL

The teaching languages developed by the Racket team for *How to Design Programs* are called Beginning Student Language, Intermediate Student Language, and Advanced Student Language. Individually they can be referred to as BSL, ISL, and ASL; collectively they are referred to as *SL. The quality of the language design bowls me over. Here is a summary:

- Data is either *atomic*, or it is *defined by parts* (product type) or *defined by choices* (sum type). ISL adds arrow types.
- There are just two syntactic categories: expression (term) and definition.
- An expression is a variable, a literal, a function application, or McCarthy's `cond`. (There are also short-circuit `and` and `or` forms.) There is no let-binding, and functions have no local variables.
- A definition introduces a function, variable, or a structure. A structure definition introduce a type predicate, a constructor function, and one selector function per field.
- A final "definition" form is `check-expect`, which is actually a unit test. There are also variations `check-within` and `check-error`.

In BSL, functions are second-class—they are not values.

ISL makes functions first class, and it adds `lambda` and `local` forms. The `local` form, like the top-level definition forms, seems to enjoy a combination of let-star and letrec semantics.

ASL adds mutation and sequencing. I haven't studied it.

Irreducible complexity: types and modules

If I want separately compiled modules with static type checking, I'm going to need a lot more syntactic forms.

What are we describing? \ What are we working with?	Values	Modules
Data	Types `type` and `datatype` definitions	Module types `module type` definitions
Computation	Values expressions `def` and `redef` definitions	Modules `module` definitions

I'll need these syntactic forms:

- *Terms* (expressions), to compute values
- *Types*, to classify terms
- *Definitions*, to associate names with terms, types, modules, and module types
- *Declarations*, to summarize definitions
- *Modules*, to collect definitions
- *Module types*, to collect declarations (and classify modules)
- *Compilation units*, to be the unit of compilation

In addition, to manage the construction of systems, I'll need these concepts:

- *Components*, to group compilation units (think CM or MLB)
- *Programs*, to be run

I'll talk loosely about a number of languages:

- The type language
- The term language
- The module language
- The component language
- An interactive language

Finally, I too plan on "language levels."

- Basic Jumbo ML is the simplest possible language, for beginners.
- Full Jumbo ML is all the shiny objects.

We'll see if I can avoid intermediate layers.

Jumbo ML

Smooth Sailing to Module Mastery

Norman Ramsey, Tufts University

The most frequently used languages are the type language and the term language. One of my goals is to make these languages look different—in Haskell, they look too much the same.

More on modules and components

Lots of discussion on modules and components. Some notes:

- *Here are the key things I want from students:*
  - *Learn to think in terms of interfaces and implementations.*
  - *Program implementations against interfaces that don't exist yet.*
  - *Implement generic data structures.*

  Greg Morrisett suggests maybe to handle all generic stuff in the core language, just by passing around extra type and value parameters. You give up some type safety, but maybe it's worth it to simplify the language.
- *I really like the units work. "Compound units" as described by Scott Owens and Matthew Flatt sound like exactly what I had in mind as components. But I need to understand why that work has both "units" and "modules."*
- *Matthias Felleisen says "don't look to units for solutions." Need to follow up.*
- *Generativity rears its ugly head everywhere. MF says it was a major issue in converting their code based to Typed Racket (do I have this right?)*
- *Wouldn't it be great to eliminate functors? MF says that units were his response to spending a year at CMU immersed in functor-land.*
- *Greg Morrisett says functors are a poor man's imitation of dependent types. Maybe there is something to be learned from recent progress in dependent types?*
- *A 2013 GPCE paper by Matthew Flatt describes Racket's "submodules," which support testing (`test`) and initialization (`main`). This looks more like a useful core-language piece of infrastructure than like something I would want to expose to students.*
- *Matthias recommends* Component Software *by Clemens Szyperski.*

For Standard ML insiders

Here's a short summary of what I've done:

- Signatures are now called "module types." Structures and functors are both just "modules."
- A `val` declaration is permitted in a module, not just a module type.
- In a polymorphic type, `forall` is always explicit.
- I've kept Standard ML's idiosyncratic convention for notating type variables, and I'm confident this is a good decision. I've also kept Standard ML's idiosyncratic way of writing type application "backwards," using postfix application—but I'm less confident that this is a good decision.
- Algebraic data types are now called "mixtures."
- In the value name space, the name of a value constructor (and only a value constructor) begins with a colon or a capital letter.
- The initial language won't have anonymous tuple or record types. When anonymous tuples and records are introduced, they'll have restrictions and a different cost semantics.
- The initial language will support pattern matching only in `case` expressions, not integrated into function definitions. And only mixtures, not structures, will participate in pattern matching.
- There's no polymorphic equality—not at any level.
- No operators are overloaded.
- Bignums will be standard, and double-precision floating-point numbers will be available.
- There's no `open`. Some of the convenience can be recovered using "`from ... import ...`."
- Legacy features like `abstype` and transparent signature ascription are gone. The `local` form is gone; let people make do with nested modules.
- Multiple arguments are handled by Currying, as in Haskell and OCaml.
- The `val` and `fun` definition forms are combined into a single new `def` form. I would also like a `redef` form.
- I want to (somehow) support mutual recursion without `and`.
- The signature language is simplified: sharing constraints and `where type` are subsumed by `&&` (least upper bound, from my ICFP'05 paper with Kathleen Fisher).
- Lexical structure is simplified: alphanumeric and symbolic characters are now on the same footing; identifiers are delimited only by whitespace, brackets, and separators (comma and semicolon). Programs are written UTF-8 and may include any character from Unicode's Basic Multilingual Plane.
- Comments are terminated by end-of-line, as in Haskell.
- There is an LL(1) grammar. The only syntactic form that extends "as far to the right as possible" is the λ form. There are curly braces and semicolons.

Jumbo ML

Smooth Sailing to Module Mastery

Norman Ramsey, Tufts University

Jumbo ML type definitions

Atomic data becomes first-class: to the student, predefined atomic data types look the same as the abstract types they define.[1] A few predefined types will be *pervasive* (a concept that is not first-class).

Product types (structures) look like this:

```
def-struct posn { x : int, y : int }
```

And we get this interface:

```
type posn
val Make-posn : int -> int -> posn
val posn-x : posn -> int
val posn-y : posn -> int
```

In the full language I would eventually like also to have

```
val Make-posn : { x : int, y : int } -> posn
```

The cost model includes heap allocation.

As in C, `def-struct` is generative.

Sum types (mixtures) look like this:

```
def-mix shape = { CIRCLE : circle -> shape
                ; SQUARE : square -> shape
                ; RECTANGLE : rectangle -> shape
                }
```

And some code:

```
val area-of-shape : shape -> double
def area-of-shape s =
  case s of
    { CIRCLE c  => Math.pi * circle-radius c * circle-radius c
    ; SQUARE sq =>
        Math.asDouble (square-side sq * square-side sq)
    ; RECTANGLE r =>
        Math.asDouble (rectangle-width r * rectangle-height r)
    }
```

As in ML, `def-mix` is generative.

[1] I'm not sure how to bootstrap an implementation.

Parametric polymorphism:

```
def-mix 'a my-list = { EMPTY : 'a my-list
                     ; CONS  : 'a -> 'a my-list -> 'a my-list
                     }

-- tycon my-list :: * => *
-- val EMPTY : forall 'a . 'a my-list
-- val CONS  : forall 'a . 'a -> 'a my-list -> 'a my-list
```

Or maybe this is better?

```
def-mix my-list -- :: * => *
   = { EMPTY : forall 'a . 'a my-list
     ; CONS  : forall 'a . 'a -> 'a my-list -> 'a my-list
     }
```

Questions:

- Does it matter that field names in structures are treated wildly inconsistently with constructor names in mixtures? (I'm unhappy with the way field names are handled in Standard ML record types, in OCaml record types, and in Haskell record notation. So I'm willing to try things that might look strange.)
- Type definitions need to communicate the types of fields, constructors, selector functions—and maybe they should also communicate the kinds of type constructors. I'm committed to writing out the types of value constructors, but not the types of selector functions. Is this OK? Reasonable? What are my choices? Is there a sweet spot?

Grammar:

def ⇒ `def-type` [*type-parameters*] *type-name* = *type*
| `def-mix` [*type-parameters*] *type-name* =
  {[;] *con-name* : *type*
  {; *con-name* [: *type*]} [;]
  }
| `def-struct` [*type-parameters*] *type-name* =
  {[;] *val-name* : *type*
  {; *val-name* : *type*} [;]
  }

Jumbo ML value definitions

def ⇒ `val` *val-name* : *type*
| `def` *val-name* [*type-parameters*] {*val-name*} = *exp*
| `redef` *val-name* [*type-parameters*] {*val-name*} = *exp*

The idea is to use `def` for everything, except when using a sequence of bindings in let-star form, to teach something resembling mutation. Technically,

- With `def`, the value being defined is visible on the right-hand side (and thus supports recursion).

Jumbo ML

Smooth Sailing to Module Mastery

Norman Ramsey, Tufts University

exp ⇒ *val-name*	Variable, e.g., `x`, `y`, ...
| *con-name*	Value constructor, e.g., `ONE`, `NOTHING`, ...
| `(`*name*`)`	Potentially infix name used as expression
| *literal*	Literal constant, e.g., `7`, `"hello"`, ...
| *...*	"Hole" that has not yet been filled in with code
| `(`λ *val-name* {*val-name*} `=>` *exp*`)`	Function, e.g., `(\ n => n + 1)`
| `[`[*exp*{`,` *exp*}]`]`	List
| *exp* {*exp*}	(Curried) function application
| *exp infix-name exp*	Infix function application
| `case` *exp* `of` {[`;`] *case-pat* `=>` *exp* {`;` *case-pat* `=>` *exp*} [`;`] `}`	Case analysis
| cond *exp* `of` {[`;`] *exp* `=>` *exp* {`;` *exp* `=>` *exp*} [`;`] `}`	Case analysis
| `local {` {*def*} `} in` *exp*	Expression with local definitions
| *exp* `:` *type*	Type assertion

Figure 1: Grammar of Jumbo ML core expressions (basic version)

- With `redef`, the value being defined is not visible on the right-hand side (and thus supports rebinding of a previously bound name).

How all this works with `val` and with mutual recursion is not clear to me—I need to look at PLT Redex and their semantics for `define` in *SL.

Greg Morrisett pointed out that it should be fine to write incomplete code with a `val` *declaration and no definition—this would correspond to defining the thing to a hole. Greg's proposal is consistent with my desire for students to be able to code against things that haven't been implemented yet.*

Unit tests, property-based testing

def ⇒ `check val` *exp* `:` *type*	
| `check` *exp* `expect` *exp* [`within` *exp*]	
| `check` *exp* `errors`	
| `check` *exp* `raises` *exp*	(Dubious)
| `check-property` [`forall` *val-name* `:` *type* {`,` *val-name* `:` *type*} `.`] *exp*	

Expression forms

See Figure 1

Declarations forms

For use in module types:

decl ⇒ [`mutable`] `type` [*type-parameters*] *type-name*	
| `type` [*type-parameters*] *type-name* `=` *type*	
| `val` *val-name* `:` *type*	
| `exn` *con-name* [`:` *type*]	
| `module` *mod-name* `:` *module-type*	
| `infix` *precedence val-name* {*val-name*}	
| `nonfix` *val-name* {*val-name*}	
| `property` [`forall` *val-name* `:` *type* {`,` *val-name* `:` *type*} `.`] *exp*	
| `property` [`forall` *val-name* `:` *type* {`,` *val-name* `:` *type*} `.`] *exp* `raises` *e*	

module-type ⇒ `{` {*decl*} `}`	
| *module-type-name*	
| *module-type* `&&` *module-type*	

def ⇒ `module type` *module-type-name* `=` *module-type*	

The Harvard audience seemed to want a lot more about properties and testing, but the only concrete suggestion I understood was from Matthias Felleisen, who wants more then just QuickCheck properties—he wants things that look like logic. I think I might be there. Matthias also says that Mike Sperber has some stuff in DMdA that does logical reasoning with properties. I hope to follow up on this.

Greg Morrisett wants a module type to be a component of a module. If so, then what is its type? I agree that managing the name space of module types would be good, but I'm not sure about the solution. Maybe only in bare modules?

Jumbo ML

Smooth Sailing to Module Mastery

Norman Ramsey, Tufts University

The tough decisions

Shall I be pure or impure? BSL and ISL are both pure languages. And supposedly one of the big lessons from Haskell was Purity is Good, Laziness Not So Much. I think the reasonable choices are as follows:

- Code is impure but there is a well-defined order of evaluation.
- All code is pure; effects are encapsulated by a parametric abstraction called the *IO monad*. The IO monad is supported by `do`-notation.

After discussion, it seems clear that purity is the way to go. For me, the deciding factor is that I would rather teach the IO monad than have to teach the value restriction. Greg Morrisett suggested that as an alternative to the IO monad, I might look for a simple type-and-effect system. This idea needs following up. Greg also says there are interesting ideas in Idris.

If code is pure, it's not clear if there should be a well-defined order of evaluation: if we leave the order of evaluation undefined, it complicates the cost model but creates opportunities for code improvement.

Another issue with pure code: we lose the ability for each module to run its own initialization code at startup. That ability is mighty handy. How should we regain it? (How do the Haskell people live without it?)

Exceptions Here are some approaches to exceptions, roughly from most libertarian to least libertarian:

- ML-style exceptions: no support in the type system; exceptions may be raised anywhere and caught anywhere; no hope of understanding large systems even with sophisticated static analysis. (Almost certainly not appropriate for Jumbo ML.)

  Exceptions can be used to signal corner cases in abstractions (e.g., least element of an empty heap), or even for arbitrary control flow.
- Liskov-style exceptions: The exceptions raised by a function are part of its type. If f calls g, then f must catch any exception raised by g, or the result is treated as fatal.

  Exceptions are used primarily to signal corner cases in abstractions.
- Haskell-style exceptions: an exception may be raised anywhere and caught anywhere on the stack, *but* the exception may be caught only in the IO monad. Because there is no defined order of evaluation, the semantics of exceptions is deliberately imprecise.

  Exceptions are used only to prevent a would-be "fatal" error from knocking out a program entirely. They are not used to signal corner cases in abstractions; that work is done using a `Maybe` (`option`) type or other sum type.
- Go-style exceptions: Used rarely to signal dire conditions; raising one must remove at least one frame from the call stack. Corner cases are signaled using "error codes."
- No exceptions at all—only checked run-time errors.

I really believed Barbara Liskov's story about the role of exceptions in abstract data types. But in practice, the real story about exceptions has not played out so well. Discuss.

Also a huge discussion here:

- *Greg Morrisett likes the maybe/option type.*
- *Matthias Felleisen likes the control operator—a modular way to connect two procedures that are far separated on the stack.*
- *Greg says it's not modular at all—too hard to reason about the state of the system when an exception is caught. Look at the proof rule.*
- *MF counters that he has used exceptions in IDEs and teachpacks—exactly where he has to interact with unknown student code that he doesn't control. I have an invitation to see some examples.*
- *Some consensus that it's a bad idea to conflate Liskov-style exceptions (least element of an empty heap) with "oh shit, I don't know what to do" exceptions.*
- *Some consensus that the real role of exceptions should be to keep the world from coming down when something goes horribly wrong. NR suggests perhaps we should be using the Erlang model instead.*

Things aren't looking good for exceptions. More notes:

- *The good thing about Liskov's worldview is that I know how to teach it to students. But maybe that worldview is obsolete?*
- *I'm not sure if the "disaster recovery" worldview actually fits in my vision for the second course.*
- *Let's not forget that Hanson is lurking in the third course.*

Mutability Should mutable data be handled with an abstraction (`ref`) as in Standard ML, or by marking some structure fields as `mutable` (as in OCaml) or the entire structure as `mutable` (cf ASL)? I lean toward making things mutable explicitly, for two reasons:

- It gives students more control over the cost model (mutability is decoupled from allocation).
- Although the language design is less parsimonious, I'm hoping it will be easier for students to learn if the idea of

mutability is present in the language and not just hidden behind an abstraction.

Composing definitions Standard ML has a very simple story about composing definitions: if d_1 and d_2 are definitions, then the sequential composition $d_1;d_2$ is also a definition. This is a story with sad consequences:

- I can't see any clean way to incorporate type signatures, e.g.,

```
val rev : forall 'a . 'a list -> 'a list
def rev xs = case xs of { NIL -> NIL
                        ; CONS x xs -> rev xs @ [x] }
```

- Mutual recursion requires the unholy `and` connective, which students consistently have trouble with.

One alternative is Haskell definitions, where everything has letrec semantics. This alternative makes less sense for an eager language.

I'd like to understand better how *SL work, because they seem to have the best of both worlds: for zero-order values, definition before use, but for functions, letrec semantics.

Modules Help! I'm drowning in a sea of modules papers. Here's what I know:

- Module types are great—they are not tied by name to an implementation, and they describe what we mean by an abstraction.
- Being able to combine abstract and manifest types in one module type is also great. Many researchers call this property *translucency*.
- To create generic, reusable modules, Standard ML uses *functors*. But specifying the arguments to a functor is hard for students: students can rarely diagnose the need for `sharing` constraints, and `where type` is a complete disaster.[2]
- I probably want to choose from this menu:
  - Classic MacQueen-style functors
  - Mixin modules
  - Flatt/Owens-style units

Unfortunately, what I know is dominated by what I don't know.

- Which of the several dozen core calculi proposed by Derek Dreyer would be a good foundation stone for a module system?
- How does one go from a core calculus to a working language design?
- Could some of the problems of classic functors be mitigated by a compiler that would, at appropriate moments, suggest sharing constraints that could be inserted?
- Why are there so many modules papers? Do I need anything from these papers beyond type soundness?
- Do I need recursive modules or units? Would they be good for students?
- Could some of these questions be answered by thinking in terms of components, not modules?

Components Most of the big, successful, statically typed languages I've worked with (Haskell, Standard ML, C) have got a notion of a thing that goes beyond the module or the compilation unit. And in all cases that thing is extralinguistic. Jumbo ML will include a language for describing *components*. Think of this language as something like a CM file or a MLton MLB file, not so much like a Cabal package description. I'm confident of the basic structure:

- A compilation unit, containing one or more modules and/or module types, is a (degenerate) component.
- The composition of a group of components is also a component.

Here are some unanswered questions:

- Within a group of components, how are import/export relations resolved?
- Does a component have a type (which presumably would say something about imports and exports)?
- How would I go about instantiating a single component in multiple ways, with different imports and exports?
- What theorems should I hope to prove about components?

***Ad hoc* polymorphism** In the glorious future, Jumbo ML will surely have some form of overloading inspired by Haskell's type classes and connected in some clean way to module types. But there is no way I am inflicting that on beginners. This position leaves me with several problems:

- In a `check ... expect ...`, how should the compiler be expected to check for equality?
- In a `property`, how should sample inputs be generated and shrunk? I know I want a clone of QuickCheck, but I also want checkable properties to be linguistic constructs, so I'll need linguistic support?
- In an interactive system, how should values be shown?
- Should arithmetic operators be overloaded?

[2] The `where type` construct uses an equals sign to equate two types, but the expressions denoting the types are elaborated in *different* environments. To say that students don't understand the need for `where type int = int` is to touch only the tip of the iceberg.

The only question I can answer is the last: no operators will be overloaded. People can import from standard modules `Int` or `Double`; Jumbo ML will support lightweight simple renaming of imported identifiers using `as`. Unlike Haskell or Modula-3, Jumbo ML won't restrict `as` to rename only modules; you'll be able to rename imported identifiers as well.

I'm not sure the rest of these issues are worth discussing—I think somebody has to do the preliminary work of developing some detailed proposals.

Candy: anonymous tuple and record types By removing anonymous tuple types, I eliminate an arbitrary decision students would otherwise have to make (should my function be Curried or tupled). By removing anonymous record types, I eliminate the whole circus around "unresolved flex records." But in the full language, I'm thinking of bringing back these constructs, but as second-class citizens:

- There are no "values" of tuple type or record type. More specifically, you can't let-bind anything of tuple or record types. If an expression on a right-hand side has a tuple or record type, then a let-binding must use pattern matching to name the individual components. The idea is a clean, cheap way to allow a function to return multiple values.
- The cost model of a tuple or record type says that no heap allocation is required: the components of a tuple or record are allocated into machine registers or on the stack.
- A tuple or record type can't be used to instantiate a type variable or to define an abstract type.
- A record type could be used as a the argument type of a function—especially a constructor function for a product type. This usage would give us named parameters.

So, mostly syntactic sugar. Does it taste sweet?

Matthias Felleisen tells me that I have reinvented "values" from Common Lisp and Scheme, and that they are a wart.

Notes

Items from discussions:

- *If we want to teach students about interfaces and implementations, we must be sure we are teaching them the solution to a problem they have actually encountered. A foundation should be laid in the first course, but still, how do we welcome them to the deep end of the pool?*
- *Matthias Felleisen teaches a 4th course called "hell," which does something similar to what Jim Waldo does in having one group use another's implementation of an agreed-upon interface—in plural.*
- *Greg Morrisett points out that ML datatypes and pattern matching militate against abstraction. Much discussion ensues. Some ideas:*
  - *Decouple "sum type" from "generative" from "recursive." One possible model is Modula-3, where recursion is the only option and generativity is handled through explicit branding.*
  - *Matthias Felleisen suggests to "take away the candy" and eliminate pattern matching entirely. I've already put substantial restrictions on pattern matching. Have to figure out more what this might look like. (It could turn out to look just like Typed Racket!)*
- *Someone suggested being able to test open terms.*
- *MF emphasizes that I have a decision to make: do I just want a new language, or do I want to create a medium in which I can easily sculpt new languages on demand. If I want a medium (and I do), Matthew Flatt is the guy to talk to. If I just want a language, anybody at Northeastern can get me started with* `#lang`.
- *MF says the two tools they have for experiments are* `#lang` *and Redex. David Van Horn has got an interesting talk/paper/demo about how to do this; look I didn't quite catch the subject matter, but something called PCF is a followup.*
- *If I understand correctly, Greg wants tuples and records to be syntactic sugar. Would like the basics to be unit, pair, and "either" types.*