Pattern Matching

> **Warning**
>
> This document was used in designing the pattern-matching features of Swift 1.0. It has not been kept to date and does not describe the current or planned behavior of Swift.

Elimination rules

When type theorists consider a programming language, we break it down like this:

- What are the kinds of fundamental and derived types in the language?
- For each type, what are its introduction rules, i.e. how do you get values of that type?
- For each type, what are its elimination rules, i.e. how do you use values of that type?

Swift has a pretty small set of types right now:

- Fundamental types: currently i1, i8, i16, i32, and i64; float and double; eventually maybe others.
- Function types.
- Tuples. Heterogeneous fixed-length products. Swift's system provides two basic kinds of element: positional and labeled.
- Arrays. Homogeneous fixed-length aggregates.
- Algebraic data types (ADTs), introduce by enum. Nominal closed disjoint unions of heterogeneous types.
- Struct types. Nominal heterogeneous fixed-length products.
- Class types. Nominal, subtypeable heterogeneous fixed-length products with identity.
- Protocol and protocol-composition types.

In addition, each of the nominal types can be made generic; this doesn't affect the overall introduction/elimination design because an "unapplied" generic type isn't first-class (intentionally), and an "applied" generic type behaves essentially like a non-generic type (also intentionally).

The point is that adding any other kind of type (e.g. SIMD vectors) means that we need to consider its intro/elim rules.

For most of these, intro rules are just a question of picking syntax, and we don't really need a document for that. So let's talk elimination. Generally, an elimination rule is a way at getting back to the information the intro rule(s) wrote into the value. So what are the specific elimination rules for these types? How do we use them, other than in type-generic ways like passing them as arguments to calls?

Functions are used by calling them. This is something of a special case: some values of function type may carry data, there isn't really a useful model for directly accessing it. Values of function type are basically completely opaque, except that we do provide thin vs. thick function types, which is potentially something we could pattern-match on, although many things can introduce thunks and so the result would not be reliable.

Scalars are used by feeding them to primitive binary operators. This is also something of a special case, because there's no useful way in which scalars can be decomposed into separate values.

Tuples, **structs**, and **classes** are used by projecting out their elements. Classes may also be turned into an object of a supertype (which is always a class).

Arrays are used by projecting out slices and elements.

Existentials are used by performing one of the operations that the type is known to support.

ADTs are used by projecting out elements of the current alternative, but how we determine the current alternative?

Alternatives for alternatives

I know of three basic designs for determining the current alternative of an ADT:

- Visitor pattern: there's some way of declaring a method on the full ADT and then implementing it for each individual alternative. You do this in OO languages mostly because there's no direct language support for closed disjoint unions (as opposed to open disjoint unions, which subclassing lets you achieve at some performance cost).
  - plus: doesn't require language support
  - plus: easy to "overload" and provide different kinds of pattern matching on the same type
  - plus: straightforward to add interesting ADT-specific logic, like matching a CallExpr instead of each of its N syntactic forms
  - plus: simple form of exhaustiveness checking
  - minus: cases are separate functions, so data and control flow is awkward
  - minus: lots of boilerplate to enable
  - minus: lots of boilerplate to use
  - minus: nested pattern matching is awful

- Query functions: dynamic_cast, dyn_cast, isa, instanceof
  - plus: easy to order and mix with other custom conditions
  - plus: low syntactic overhead for testing the alternative if you don't need to actually decompose
  - minus: higher syntactic overhead for decomposition
    - isa/instanceof pattern requires either a separate cast or unsafe operations later
    - dyn_cast pattern needs a fresh variable declaration, which is very awkward in complex conditions
  - minus: exhaustiveness checking is basically out the window
  - minus: some amount of boilerplate to enable
- Pattern matching
  - plus: no boilerplate to enable
  - plus: hugely reduced syntax to use if you want a full decomposition
  - plus: compiler-supported exhaustiveness checking
  - plus: nested matching is natural
  - plus: with pattern guards, natural mixing of custom conditions
  - minus: syntactic overkill to just test for a specific alternative (e.g. to filter it out)
  - minus: needs boilerplate to project out a common member across multiple/all alternatives
  - minus: awkward to group alternatives (fallthrough is a simple option but has issues)
  - minus: traditionally completely autogenerated by compiler and thus not very flexible
  - minus: usually a new grammar production that's very ambiguous with the expression grammar
  - minus: somewhat fragile against adding extra data to an alternative

I feel that this strongly points towards using pattern matching as the basic way of consuming ADTs, maybe with special dispensations for querying the alternative and projecting out common members.

Pattern matching was probably a foregone conclusion, but I wanted to spell out that having ADTs in the language is what really forces our hand because the alternatives are so bad. Once we need pattern-matching, it makes sense to provide patterns for the other kinds of types as well.

Selection statement

This is the main way we expect users to employ non-obvious pattern-matching. We obviously need something with statement children, so this has to be a statement. That's also fine because this kind of full pattern match is very syntactically heavyweight, and nobody would want to embed it in the middle of an expression. We also want a low-weight matching expression, though, for relatively simple ADTs:

```
stmt            ::= stmt-switch
stmt-switch     ::= 'switch' expr '{' switch-group+ '}'
switch-group    ::= case-introducer+ stmt-brace-item+
case-introducer ::= 'case' match-pattern-list case-guard? ':'
case-introducer ::= 'default' case-guard? ':'
case-guard      ::= 'where' expr
match-pattern-list ::= match-pattern
match-pattern-list ::= match-pattern-list ',' match-pattern
```

We can get away with using "switch" here because we're going to unify both values and patterns under match-pattern. The works chiefly by making decompositional binding a bit more awkward, but has the major upside of reducing the likelihood of dumb mistakes (rebinding 'true', for example), and it means that C-looking switches actually match our semantics quite closely. The latter is something of a priority: a C switch over an enum is actually pretty elegant --- well, except for all the explicit scoping and 'break' statements, but the switching side of it feels clean.

Default

I keep going back and forth about having a "default" case-introducer. On the one hand, I kind of want to encourage total matches. On the other hand, (1) having it is consistent with C, (2) it's not an unnatural style, and (3) there are cases where exhaustive switching isn't going to be possible. We can certainly recommend complete matches in switches, though.

If we do have a 'default', I think it makes the most sense for it to be semantically a complete match and therefore require it to be positioned at the end (on pain of later matches being irrelevant). First, this gives more sensible behavior to 'default where x.isPurple()', which really doesn't seem like it should get reordered with the surrounding cases; and second, it makes the matching story very straightforward. And users who like to put 'default:' at the top won't accidentally get unexpected behavior because coverage checking will immediately complain about the fact that every case after an unguarded 'default' is obviously dead.

Case groups

A case-group lets you do the same thing for multiple cases without an extra syntactic overhead (like a 'fallthrough' after every case). For some types (e.g. classic functional linked lists) this is basically pointless, but for a lot of other types (Int, enums, etc.) it's pervasive.

The most important semantic design point here is about bound variables in a grouped case, e.g. (using 'var' as a "bind this variable" introducer; see the pattern grammar):

```
switch (pair) {
case (var x, 0):
case (0, var y):
  return 1
case (var x, var y)
  return foo(x - 1, y) + foo(x, y - 1)
}
```

It's tempting to just say that an unsound name binding (i.e. a name not bound in all cases or bound to values of different types) is just always an error, but I think that's probably not the way to go. There are two things I have in mind here: first, these variables can be useful in pattern guards even if they're not used in the case block itself, and second, a well-chosen name can make a pattern much more self-documenting. So I think it should only be an error to *refer* to an unsound name binding.

The most important syntactic design point here is whether to require (or even allow) the 'case' keyword to be repeated for each case. In many cases, it can be much more compact to allow a comma-separated list of patterns after 'case':

```
switch (day) {
case .Terrible, .Horrible, .NoGood, .VeryBad:
  abort()
case .ActuallyPrettyReasonableWhenYouLookBackOnIt:
  continue
}
```

or even more so:

```
case 0...2, 5...10, 14...18, 22...:
  flagConditionallyAcceptableAge()
```

On the other hand, if this list gets really long, the wrapping gets a little weird:

```
case .Terrible, .Horrible, .NoGood, .VeryBad,
     .Awful, .Dreadful, .Appalling, .Horrendous,
     .Deplorable, .Unpleasant, .Ghastly, .Dire:
  abort()
```

And while I think pattern guards should be able to apply to multiple cases, it would be nice to allow different cases in a group to have different pattern guards:

```
case .None:
case .Some(var c) where c.isSpace() || c.isASCIIControl():
  skipToEOL()
```

So really I think we should permit multiple 'case' introducers:

```
case .Terrible, .Horrible, .NoGood, .VeryBad:
case .Awful, .Dreadful, .Appalling, .Horrendous:
case .Deplorable, .Unpleasant, .Ghastly, .Dire:
  abort()
```

With the rule that a pattern guard can only use bindings that are sound across its guarded patterns (those within the same 'case'), and the statement itself can only use bindings that are sound across all of the cases. A reference that refers to an unsound binding is an error; lookup doesn't just ignore the binding.

Scoping

Despite the lack of grouping braces, the semantics are that the statements in each case-group form their own scope, and falling off the end causes control to resume at the end of the switch statement -- i.e. "implicit break", not "implicit fallthrough".

Chris seems motivated to eventually add an explicit 'fallthrough' statement. If we did this, my preference would be to generalize it by allowing the match to be performed again with a new value, e.g. `fallthrough(something)`, at least optionally. I think having local functions removes a lot of the impetus, but not so much as to render the feature worthless.

Syntactically, braces and the choice of case keywords are all bound together. The thinking goes as follows. In Swift, statement scopes are always grouped by braces. It's natural to group the cases with braces as well. Doing both lets us avoid a 'case' keyword, but otherwise it leads to ugly style, because either the last case ends in two braces on the same line or cases have to further indented. Okay, it's easy enough to not require braces on the match, with the grammar saying that cases are just greedily consumed -- there's no ambiguity here because the switch statement is necessarily within braces. But that leaves the code without a definitive end to the cases, and the closing braces end up causing a lot of unnecessary vertical whitespace, like so:

```
switch (x)
case .foo {
  // ...
}
case .bar {
  // ...
}
```

So instead, let's require the switch statement to have braces, and we'll allow the cases to be written without them:

```
switch (x) {
case .foo:
  // ...
case .bar:
  // ...
}
```

That's really a lot prettier, except it breaks the rule about always grouping scopes with braces (we *definitely* want different cases to establish different scopes). Something has to give, though.

We require the trailing colon because it's a huge cue for separating things, really making single-line cases visually appealing, and the fact that it doesn't suggest closing punctuation is a huge boon. It's also directly precedented in C, and it's even roughly the right grammatical function.

Case selection semantics

The semantics of a switch statement are to first evaluate the value operand, then proceed down the list of case-introducers and execute the statements for the switch-group that had the first satisfied introducer.

It is an error if a case-pattern can never trigger because earlier cases are exhaustive. Some kinds of pattern (like 'default' cases and '_') are obviously exhaustive by themselves, but other patterns (like patterns on properties) can be much harder to reason about exhaustiveness for, and of course pattern guards can make this outright undecidable. It may be easiest to apply very straightforward rules (like "ignore guarded patterns") for the purposes of deciding whether the program is actually ill-formed; anything else that we can prove is unreachable would only merit a warning. We'll probably also want a way to say explicitly that a case can never occur (with semantics like llvm_unreachable, i.e. a reliable runtime failure unless that kind of runtime safety checking is disabled at compile-time).

A 'default' is satisfied if it has no guard or if the guard evaluates to true.

A 'case' is satisfied if the pattern is satisfied and, if there's a guard, the guard evaluates to true after binding variables. The guard is not evaluated if the pattern is not fully satisfied. We'll talk about satisfying a pattern later.

Non-exhaustive switches

Since falling out of a statement is reasonable behavior in an imperative language -- in contrast to, say, a functional language where you're in an expression and you need to produce a value -- there's a colorable argument that non-exhaustive matches should be okay. I dislike this, however, and propose that it should be an error to make a non-exhaustive switch; people who want non-exhaustive matches can explicitly put in default cases. Exhaustiveness actually isn't that difficult to check, at least over ADTs. It's also really the behavior that I would expect from the syntax, or at least implicitly falling out seems dangerous in a way that nonexhaustive checking doesn't. The complications with checking exhaustiveness are pattern guards and matching expressions. The obvious conservatively-safe rule is to say "ignore cases with pattern guards or matching expressions during exhaustiveness checking", but some people really want to write "where x < 10" and "where x >= 10", and I can see their point. At the same time, we really don't want to go down that road.

Other uses of patterns

Patterns come up (or could potentially come up) in a few other places in the grammar:

Var bindings

Variable bindings only have a single pattern, which has to be exhaustive, which also means there's no point in supporting guards here. I think we just get this:

```
decl-var ::= 'var' attribute-list? pattern-exhaustive value-specifier
```

Function parameters

The functional languages all permit you to directly pattern-match in the function declaration, like this example from SML:

```
fun length nil = 0
  | length (a::b) = 1 + length b
```

This is really convenient, but there's probably no reasonable analogue in Swift. One specific reason: we want functions to be callable with keyword arguments, but if you don't give all the parameters their own names, that won't work.

The current Swift approximation is:

```
func length(_ list : List) : Int {
  switch list {
    case .nil: return 0
    case .cons(_, var tail): return 1 + length(tail)
  }
}
```

That's quite a bit more syntax, but it's mostly the extra braces from the function body. We could remove those with something like

this:

```
func length(_ list : List) : Int = switch list {
  case .nil: return 0
  case .cons(_, var tail): return 1 + length(tail)
}
```

Anyway, that's easy to add later if we see the need.

Assignment

This is a bit iffy. It's a lot like var bindings, but it doesn't have a keyword, so it's really kind of ambiguous given the pattern grammar.

Also, l-value patterns are weird. I can come up with semantics for this, but I don't know what the neighbors will think:

```
var perimeter : double
.feet(x) += yard.dimensions.height // returns Feet, which has one constructor, :feet.
.feet(x) += yard.dimensions.width
```

It's probably better to just have l-value tuple expressions and not try to work in arbitrary patterns.

Pattern-match expression

This is an attempt to provide that dispensation for query functions we were talking about.

I think this should bind looser than any binary operators except assignments; effectively we should have:

```
expr-binary ::= # most of the current expr grammar

expr ::= expr-binary
expr ::= expr-binary 'is' expr-primary pattern-guard?
```

The semantics are that this evaluates to true if the pattern and pattern-guard are satisfied.

'is' or 'isa'

Perl and Ruby use '=~' as the regexp pattern-matching operator, which is both obscure and really looks like an assignment operator, so I'm stealing Joe's 'is' operator, which is currently used for dynamic type-checks. I'm of two minds about this: I like 'is' a lot for value-matching, but not for dynamic type-checks.

One possibility would be to use 'is' as the generic pattern-matching operator but use a different spelling (like 'isa') for dynamic type-checks, including the 'is' pattern. This would give us "x isa NSObject" as an expression and "case isa NSObject:" as a case selector, both of which I feel read much better. But in this proposal, we just use a single operator.

Other alternatives to 'is' include 'matches' (reads very naturally but is somewhat verbose) or some sort of novel operator like '~~'.

Note that this impacts a discussion in the section below about expression patterns.

Dominance

I think that this feature is far more powerful if the name bindings, type-refinements, etc. from patterns are available in code for which a trivial analysis would reveal that the result of the expression is true. For example:

```
if s is Window where x.isVisible {
  // can use Window methods on x here
}
```

Taken a bit further, we can remove the need for 'where' in the expression form:

```
if x is Window && x.isVisible { ... }
```

That might be problematic without hard-coding the common control-flow operators, though. (As well as hardcoding some assumptions about Bool.convertToLogicValue...)

Pattern grammar

The usual syntax rule from functional languages is that the pattern grammar mirrors the introduction-rule expression grammar, but parses a pattern wherever you would otherwise put an expression. This means that, for example, if we add array literal expressions, we should also add a corresponding array literal pattern. I think that principle is very natural and worth sticking to wherever possible.

Two kinds of pattern

We're blurring the distinction between patterns and expressions a lot here. My current thinking is that this simplifies things for the programmer --- the user concept becomes basically "check whether we're equal to this expression, but allow some holes and some more complex 'matcher' values". But it's possible that it instead might be really badly confusing. We'll see! It'll be fun!

This kind of forces us to have parallel pattern grammars for the two major clients:

- Match patterns are used in `switch` and `matches`, where we're decomposing something with a real possibility of failing. This means that expressions are okay in leaf positions, but that name-bindings need to be explicitly advertised in some way to reasonably disambiguate them from expressions.
- Exhaustive patterns are used in `var` declarations and function signatures. They're not allowed to be non-exhaustive, so having a match expression doesn't make any sense. Name bindings are common and so shouldn't be penalized.

You might think that having a "pattern" as basic as `foo` mean something different in two different contexts would be confusing, but actually I don't think people will generally think of these as the same production -- you might if you were in a functional language where you really can decompose in a function signature, but we don't allow that, and I think that will serve to divide them in programmers' minds. So we can get away with some things. :)

Binding patterns

In general, a lot of these productions are the same, so I'm going to talk about `*`-patterns, with some specific special rules that only apply to specific pattern kinds.

```
*-pattern ::= '_'
```

A single-underscore identifier is always an "ignore" pattern. It matches anything, but does not bind it to a variable.

```
exhaustive-pattern ::= identifier
match-pattern ::= '?' identifier
```

Any more complicated identifier is a variable-binding pattern. It is illegal to bind the same identifier multiple times within a pattern. However, the variable does come into scope immediately, so in a match pattern you can have a latter expression which refers to an already-bound variable. I'm comfortable with constraining this to only work "conveniently" left-to-right and requiring more complicated matches to use guard expressions.

In a match pattern, variable bindings must be prefixed with a ? to disambiguate them from an expression consisting of a variable reference. I considered using 'var' instead, but using punctuation means we don't need a space, which means this is much more compact in practice.

Annotation patterns

```
exhaustive-pattern ::= exhaustive-pattern ':' type
```

In an exhaustive pattern, you can annotate an arbitrary sub-pattern with a type. This is useful in an exhaustive pattern: the type of a variable isn't always inferable (or correctly inferable), and types in function signatures are generally outright required. It's not as useful in a match pattern, and the colon can be grammatically awkward there, so we disallow it.

'is' patterns

```
match-pattern ::= 'is' type
```

This pattern is satisfied if the dynamic type of the matched value "satisfies" the named type:

- if the named type is an Objective-C class type, the dynamic type must be a class type, and an 'isKindOf:' check is performed;
- if the named type is a Swift class type, the dynamic type must be a class type, and a subtype check is performed;
- if the named type is a metatype, the dynamic type must be a metatype, and the object type of the dynamic type must satisfy the object type of the named type;
- otherwise the named type must equal the dynamic type.

This inquiry is about dynamic types; archetypes and existentials are looked through.

The pattern is ill-formed if it provably cannot be satisfied.

In a 'switch' statement, this would typically appear like this:

```
case is NSObject:
```

It can, however, appear in recursive positions:

```
case (is NSObject, is NSObject):
```

Ambiguity with type value matching

There is a potential point of confusion here with dynamic type checking (done by an 'is' pattern) vs. value equality on type objects (done by an expression pattern where the expression is of metatype type. This is resolved by the proposal (currently outstanding but generally accepted, I think) to disallow naked references to type constants and instead require them to be somehow decorated.

That is, this pattern requires the user to write something like this:

```
case is NSObject:
```

It is quite likely that users will often accidentally write something like this:

```
case NSObject:
```

It would be very bad if that were actually accepted as a valid expression but with the very different semantics of testing equality of type objects. For the most part, type-checking would reject that as invalid, but a switch on (say) a value of archetype type would generally work around that.

However, we have an outstanding proposal to generally forbid 'NSObject' from appearing as a general expression; the user would have to decorate it like the following, which would let us eliminate the common mistake:

```
case NSObject.type:
```

Type refinement

If the value matched is immediately the value of a local variable, I think it would be really useful if this pattern could introduce a type refinement within its case, so that the local variable would have the refined type within that scope. However, making this kind of type refinement sound would require us to prevent there from being any sort of mutable alias of the local variable under an unrefined type. That's usually going to be fine in Swift because we usually don't permit the address of a local to escape in a way that crosses statement boundaries. However, closures are a major problem for this model. If we had immutable local bindings --- and, better yet, if they were the default --- this problem would largely go away.

This sort of type refinement could also be a problem with code like:

```
while expr is ParenExpr {
  expr = expr.getSubExpr()
}
```

It's tricky.

"Call" patterns

```
match-pattern ::= match-pattern-identifier match-pattern-tuple?
match-pattern-identifier ::= '.' identifier
match-pattern-identifier ::= match-pattern-identifier-tower
match-pattern-identifier-tower ::= identifier
match-pattern-identifier-tower ::= identifier
match-pattern-identifier-tower ::= match-pattern-identifier-tower '.' identifier
```

A match pattern can resemble a global name or a call to a global name. The global name is resolved as normal, and then the pattern is interpreted according to what is found:

- If the name resolves to a type, then the dynamic type of the matched value must match the named type (according to the rules below for 'is' patterns). It is okay for this to be trivially true.

  In addition, there must be a non-empty arguments clause, and each element in the clause must have an identifier. For each element, the identifier must correspond to a known property of the named type, and the value of that property must satisfy the element pattern.

- If the name resolves to an enum element, then the dynamic type of the matched value must match the enum type as discussed above, and the value must be of the specified element. There must be an arguments clause if and only if the element has a value type. If so, the value of the element is matched against the clause pattern.
- Otherwise, the argument clause (if present) must also be syntactically valid as an expression, and the entire pattern is reinterpreted as an expression.

This is all a bit lookup-sensitive, which makes me uncomfortable, but otherwise I think it makes for attractive syntax. I'm also a little worried about the way that, say, `f(x)` is always an expression but `A(x)` is a pattern. Requiring property names when matching properties goes some way towards making that okay.

I'm not totally sold on not allowing positional matching against struct elements; that seems unfortunate in cases where positionality is conventionally unambiguous, like with a point.

Matching against struct types requires arguments because this is intended to be used for structure decomposition, not dynamic type testing. For the latter, an 'is' pattern should be used.

Expression patterns

```
match-pattern ::= expression
```

When ambiguous, match patterns are interpreted using a pattern-specific production. I believe it should be true that, in general, match patterns for a production accept a strict superset of valid expressions, so that (e.g.) we do not need to disambiguate whether an open paren starts a tuple expression or a tuple pattern, but can instead just aggressively parse as a pattern. Note that binary operators can mean that, using this strategy, we sometimes have to retroactively rewrite a pattern as an expression.

It's always possible to disambiguate something as an expression by doing something not allowing in patterns, like using a unary operator or calling an identity function; those seem like unfortunate language solutions, though.

Satisfying an expression pattern

A value satisfies an expression pattern if the match operation succeeds. I think it would be natural for this match operation to be spelled the same way as that match-expression operator, so e.g. a member function called 'matches' or a global binary operator called '~' or whatever.

The lookup of this operation poses some interesting questions. In general, the operation itself is likely to be associated with the intended type of the expression pattern, but that type will often require refinement from the type of the matched value.

For example, consider a pattern like this:

```
case 0...10:
```

We should be able to use this pattern when switching on a value which is not an Int, but if we type-check the expression on its own, we will assign it the type Range<Int>, which will not necessarily permit us to match (say) a UInt8.

Order of evaluation of patterns

I'd like to keep the order of evaluation and testing of expressions within a pattern unspecified if I can; I imagine that there should be a lot of cases where we can rule out a case using a cheap test instead of a more expensive one, and it would suck to have to run the expensive one just to have cleaner formal semantics. Specifically, I'm worried about cases like `case [foo(), 0]:`; if we can test against 0 before calling `foo()`, that would be great. Also, if a name is bound and then used directly as an expression later on, it would be nice to have some flexibility about which value is actually copied into the variable, but this is less critical.

```
*-pattern ::= *-pattern-tuple
*-pattern-tuple ::= '(' *-pattern-tuple-element-list? '...'? ')'
*-pattern-tuple-element-list ::= *-pattern-tuple-element
*-pattern-tuple-element-list ::= *-pattern-tuple-element ',' pattern-tuple-element-list
*-pattern-tuple-element ::= *-pattern
*-pattern-tuple-element ::= identifier '=' *-pattern
```

Tuples are interesting because of the labeled / non-labeled distinction. Especially with labeled elements, it is really nice to be able to ignore all the elements you don't care about. This grammar permits some prefix or set of labels to be matched and the rest to be ignored.

Miscellaneous

It would be interesting to allow overloading / customization of pattern-matching. We may find ourselves needing to do something like this to support non-fragile pattern matching anyway (if there's some set of restrictions that make it reasonable to permit that). The obvious idea of compiling into the visitor pattern is a bit compelling, although control flow would be tricky -- we'd probably need the generated code to throw an exception. Alternatively, we could let the non-fragile type convert itself into a fragile type for purposes of pattern matching.

If we ever allow infix ADT constructors, we'll need to allow them in patterns as well.

Eventually, we will build regular expressions into the language, and we will allow them directly as patterns and even bind grouping expressions into user variables.

John.