Pattern Matching

Introducing Identifiers

R0: Annotate *id-expr* with ^

```
inspect (e) {
  name => ...
  0 => ...
  ^value => ...

  [x, y] => ...
  [0, ^a] => ...
  [^a, ^b] => ...
  [x, y, z, ^a] => // more names
  [^a, ^b, ^c, x] => // more refs

  <Circle> circle => ...
  <Rectangle> [width, height] => ...
};
```

```
// Pin (^) operator in Elixir

// Weird looking simple uses
enum Color { Red, Blue };

inspect (color) {
  ^Red => ...
  ^Blue => ...
};

// Breaks MSFT smart pointer extension?
```

R1/R2: let/case recursive, let default

```
inspect (e) {
  name => ...
  0 => ...
  case value => ...

  [x, y] => ...
  [0, case b] => ...
  case [a, b] => ...
  [x, y, z, case a] => // more names
  case [a, b, c, let x] => // more refs

  <Circle> circle => ...
  <Rectangle> [width, height] => ...
};
```

```
// let from Rust, Swift, many others

// + `switch` looking simple uses
enum Color { Red, Blue };

inspect (color) {
  case Red => ...
  case Blue => ...
};

// - Recursing gets complex with nesting
inspect (e) {
  let [a, case [let b, c], [d]] => ...
};
```

Cologne 2019

1. P1371R1 We support the authors' direction of 'let' and 'case' modes applying to subpatterns

SF	F	N	A	SA
7	7	5	4	0

R3: Annotate *id-expr* with case

```
inspect (e) {
  name => ...
  0 => ...
  case value => ...

  [x, y] => ...
  [0, case b] => ...
  [case a, case b] => ...
  [x, y, z, case a] => // more names
  [case a, case b, case c, x] => // refs

  <Circle> circle => ...
  <Rectangle> [width, height] => ...
};
```

```
// + `switch` looking simple uses
enum Color { Red, Blue };

inspect (color) {
  case Red => ...
  case Blue => ...
};

// + Easier to read in nested patterns
inspect (e) {
  [a, [b, case c], [d]] => ...
};

// + No need to introduce let.
// - Abusing case
// - Expressions should be expressions
// - Declaration of names should look
//   more like a declaration
//   (lambda captures were a mistake?)
```

R4? Annotate *id-pat* instead?

```
inspect (e) {
  let name => ...
  0 => ...
  value => ...

  [let x, let y] => ...
  [0, b] => ...
  [a, b] => ...
  [let x, let y, let z, a] => // names
  [a, b, c, let x] => // more refs

  <Circle> let circle => ...
  <Rectangle> [let width, let height] => ...
};
```

```
// Clean simple use cases
enum Color { Red, Blue };

inspect (color) {
  Red => ...
  Blue => ...
};

// Easier to read through deep nesting
inspect (e) {
  [let a, [let b, c], [let d]] => ...
};

// + No longer abusing case
// + Expressions are expressions
// + Declaration of names have
//   an indication of a declaration.
// - Apparent inconsistency with
//   structured bindings.
// * No longer optimizing for what
//   some believe is more common.
```

Diagnostics

```
// name introduces id

int x = 0;

inspect (e) {
  foo => // introduce foo,
         // works as expected 👍

  x => // introduce x,
       // works as expected 👍

  x => // wanted to match value,
       // ends up shadowing.
       // likely no diagnostics 👎
};
```

```
// name refers to existing id

int x = 0;

inspect (e) {
  foo => // wanted to introduce foo,
         // look-up fails and error 👍

  x => // wanted to introduce x,
       // ends up with matching.
       // the name accidentally chosen
       // need be comparable to `e`.
       // if not, error

  x => // match value,
       // works as expected 👍
};
```

Structured Bindings

```
auto [x, y] = e;
// ...
```

```
inspect (e) {
  [let x, let y] => ...
};
```

A Mixed Context

Expressions perform name lookup on existing identifiers.
Examples: `x, f(x), x && y`

Declarations introduce new identifiers.
Examples: `int x; void f(int x); auto [x, y] = e;`

Patterns want to do some of both!
Examples: `x, let x, [x, y], [let x, y]`

let vs auto

```
inspect (e) {
  auto name => // &name != &e
  auto&& name => // &name == &e
  0 => ...
  value => ...

  auto&& [x, y] => // x and y are bindings
  [0, b] => ...
  [a, b] => ...
  [auto&& x, auto&& y, auto&& z, a] => // more names
  [a, b, c, auto&& x] => // more refs

  <Circle> auto&& c => ...
  <Rectangle> auto&& [width, height] => ...
};
```

auto name

```
struct S {
  S() = default;
  S(const S&) = delete;
  S(S&&) = delete;
} s;

inspect (s) {
  auto name => ...
  __ => ...
};

// Is this a non-match, or a compilation error?
```

auto&& name

```
struct S {} s;

inspect (s) {
  auto&& name => ...
};

// Vast majority of the uses will likely be auto&& to avoid copies.
// Given these familiar set of choices, many may opt for auto const&

inspect (pair) {
  [auto const& first, auto const& second] => // mouthful
};
```

Structured Bindings

```
inspect (e) {
  0 => ...
  value => ...

  auto&& [x, y] => // x and y are bindings

  auto&& [x, 0] => // disallowed?

  [auto&& x, 0] => // okay
  [auto&& x, value] => // okay
};
```

Structured Bindings

```
inspect (e) {
  auto&& [x, y] => // x and y are bindings

  [auto&& x, 0] => // x is not a binding, cannot bind to bitfields.
};

// Extend references to bind bitfields? only auto&& ?
// Is this actually feasible?
```

Reification of bindings

```
inspect (e) {
  [let x, let y] => // x and y are bindings

  [let x, 0] => // x is a binding, can bind to bitfields.
};
```

Lifetimes

```
// Structured bindings considers bindings to the components essential,
// but not the lifetimes of each component.
//
// However, it does allow control of the entire object's lifetime
// rather than always declaring it as `auto&&`. -- Why?

   struct S {
     Data const& data() const { return data_; }
     Data data_;
   };

   S f();

   auto&& [x, y] = f().data();  // similar to for (auto& x : f().data())
// ^^^^^^ this is dangling.

   auto [x, y] = f().data(); // okay
```

Lifetimes

```
// Such lifetime problems are not present in pattern matching.

   struct S {
     Data const& data() const { return data_; }
     Data data_;
   };

   S f();

   auto&& [x, y] = f().data();
// ^^^^^^ this is dangling.

   inspect (f().data()) {  // this is safe!
     /* bindings here are also safe */ => ...
   };
```

let vs auto

```
inspect (e) {
  let name => ...
  0 => ...
  value => ...

  [let x, let y] => ...
  [0, b] => ...
  [a, b] => ...
  [let x, let y, let z, a] => // more names
  [a, b, c, let x] => // more refs

  <Circle> let circle => ...
  <Rectangle> [let width, let height] => ...
};

// - new context-sensitive keyword `let`
// + simpler, - no lifetime control
// + no unintentional copies
// + less verbose
// + explicit spelling of bindings
```

```
inspect (e) {
  auto name => ...
  0 => ...
  value => ...

  auto&& [x, y] => ...
  [0, b] => ...
  [a, b] => ...
  [auto x, auto const& y, auto&& z, a] => ...
  [a, b, c, auto&& x] => // more refs

  <Circle> auto&& c => ...
  <Rectangle> auto&& [width, height] => ...
};

// + familiar syntax, no new keyword
// - complexity, + finer lifetime control
// - potential unintentional copies
// - more verbose
// - bindings are all or nothing, tied to SB
```

let/case

```
inspect (e) {
  name => ...
  0 => ...
  case value => ...

  [x, y] => ...
  [0, case b] => ...
  [case a, case b] => ...
  let [x, y, case a] =>
  case [a, b, let x] =>

  <Circle> let circle => ...
  <Rect> let [w, h] =>
};
```

let

```
inspect (e) {
  let name => ...
  0 => ...
  value => ...

  let [x, y] => ...
  [0, b] => ...
  [a, b] => ...
  [let x, let y, a] =>
  [a, b, let x] =>

  <Circle> let circle => ...
  <Rect> let [w, h] =>
};
```

auto

```
inspect (e) {
  auto name => ...
  0 => ...
  value => ...

  auto&& [x, y] => ...
  [0, b] => ...
  [a, b] => ...
  [auto x, auto&& y, a] =>
  [a, b, auto&& x] =>

  <Circle> auto&& c => ...
  <Rect> auto&& [w, h] =>
};
```