Pattern matching syntax analysis

Key
Test clause
Assignment
Both test and assignment
Object construction
Alias
Both test and Alias

Status quo:

```
let res = { status: 200 }

let status = 200
let { status } = res
let { status : stat } = res
```

Example 1

```
match (res) {
  when ({ status: 200, body, ...rest }):
    handleData(body, rest)
  when ({ status, destination: let url })
      if (300 <= status && status < 400):
    handleRedirect(url)
  when ({ status: 500 }) if (!this.hasRetried): do {
    retry(req);
    this.hasRetried = true;
  }
  default: throwSomething();
}
```

To comprehend by reading: you need to first determine: what is it checking for? What is it assigning? What is it aliasing? This will take multiple read throughs, with additional rules and clauses to keep in mind. That is 3 readings for 1 line. This results in a **heavy mental load**. In addition -- the tools at our disposal to correct this via syntax highlighting are limited to non-existent because of this multi-meaning layering, consider -- if we have a shadow variable `url` in scope, how does the tool decide if destination is an alias or a match?. That means we cannot correct this heavy mental load easily.

Also, consider if we make one small change to the code:

```
let url = “www.hi.com”

match (res) {
  when ({ status: 200, body, ...rest }):
    handleData(body, rest)
  when ({ status, destination: url })
      if (300 <= status && status < 400):
    handleRedirect(url)
  when ({ status: 500 }) if (!this.hasRetried): do {
    retry(req);
    this.hasRetried = true;
  }
  default: throwSomething();
}
```

What is going to happen to the alias of destination now? What is the programmers intention?

In addition, we are rewriting the rules of what destructuring does. This means that the entire language becomes burdened with additional rules, for something that already relies on a lot of implicit knowledge (has an inherently heavy mental load). The result is poor learnability -- we can not transfer concepts we have learned before to new code.

Let’s rewrite that. Consider this alternative:

```
function maybeRetry(res) {
  // for more complex cases, we can use functions which return
  // a boolean.
  return res.status == 500 && !this.hasRetried;
}

match (res) {
  let { status, body, ...rest } when { status: 200, body }:
    handleData(body, rest)
  let { destination: url }
        when { status and status >= 300 and status < 400, destination
}:
    handleRedirect(url)
  when maybeRetry.bind(this): { // can also be a higher order fn
     retry(req);
```

```
      this.hasRetried = true;
  }
  default: throwSomething();
}
```

I've retained the initial duplication of variable bindings intentionally here. I know this will be a criticism, but I will address that at the end.

In this example, we have split assignment/aliasing and conditional testing. The result is you read the line once. It has a lighter mental load due to only one layer of information. In addition, the lookup here is fast: you can orient your eyes to the left or the right on the `when`. Finally, this follows pre-existing rules on object destructuring: the left hand side works the same way it always has. The right hand side is somewhat new, but not as new as in the previous example.

Example 2

```
match (command) {
  when (['go', dir and ('north' or 'east' or 'south' or 'west')]):
    go(dir);
  when (['take', item and /[a-z]+ ball/ and { weight }]):
    take(item);
  default: lookAround()
}
```

Let's rewrite that. Consider this alternative:

```
match (command) {
  let [_, dir] when ['go', ('north' or 'east' or 'south' or 'west')]:
go(dir);
  let [_, item] when ['take', (/[a-z]+ ball/ and { weight })]:
take(item);
  default: lookAround();
}

Number[@@Symbol.matcher]

Let foo :: Number = 1

match (command) {
  when { status: let s, url :: "h..." }: go(dir);
```

```
  when [ status ] : take(item);
  default: lookAround();
}

status [@@Symbol.matcher]

Function foo(matchee) {
  Return {
   Match: true
   Value: ...
  }
}

let x when (x) = y

Let x = functionblah.match(whatever) ? kjsdhfjksh : ….

Regex.match
customMatcher.match

when [ status ] <-- irrefutable match, bound to 'status'
when [ status as s :: ] <-- irrefutable match, bound to s, a bit
absurd
when [ status :: * ] <-- irrefutable match, bound to 'status'
when [ status :: foo ] <-- custom match, bound to 'status'
when [ status :: 500 ] <-- custom match, bound to 'status'


when [ { err: { cause: { status }}} as wrappedErr ]

when [ { err :: { cause :: { status }}} as wrappedErr ]
when [ { err :: { cause :: HttpError }} as wrappedErr ]
when [ HttpError ]

match (command) {
  let [_, dir] when ['go', ('north' or 'east' or 'south' or 'west')]:
go(dir);
```

```
  let [_, item] when ['take', (/[a-z]+ ball/ and { weight })]:
take(item);
  default: lookAround();
}
```

These look almost identical and are almost exactly the same in terms of character length. However one has a simple visual rule separating assignment and testing.

Example 3

```
match (result) {
  when (${Option.Some} with val): console.log(val);
  when (${Option.None}): console.log("none");
}
```

Note that the assignment position is inverted from normal assignment.

Lets rewrite that. Consider this alternative:

```
match (result) {
  let val when Option.isSome: console.log(val);
  when Option.isNone: console.log("none");
}
```

Here, the assignment is in its default position: the left hand side.

What is important to note, is how this compares to the previous example, which _had_ assignment. Consider these together:

Compound example:

```
match (res) {
  when ({ status: 200, body, ...rest }):
    handleData(body, rest)
  when ({ status, destination: url })
      if (300 <= status && status < 400):
    handleRedirect(url)
  when ({ status: 500 }) if (!this.hasRetried): do {
    retry(req);
    this.hasRetried = true;
```

```
  }
  default: throwSomething();
}

match (command) {
  when (['go', dir and ('north' or 'east' or 'south' or 'west')]):
    go(dir);
  when (['take', item and /[a-z]+ ball/ and { weight }]):
    take(item);
  default: lookAround()
}
```

The two match statements here end up being read differently because of the data structure. This means that searching for the assigned value will be difficult. This will particularly hurt if we take the first case and rewrite it as an array data structure:

```
match (res) {
  when ({ status: 200, body, ...rest }):
    handleData(body, rest)
  when ({ status, destination: url })
      if (300 <= status && status < 400):
    handleRedirect(url)
  when ({ status: 500 }) if (!this.hasRetried): do {
    retry(req);
    this.hasRetried = true;
  }
  default: throwSomething();
}

match (res) {
  when ([ 200, body, ...rest ]):
    handleData(body, rest)
  when ([ status, _, url ])
      if (300 <= res[0] && res[0] < 400):
    handleRedirect(url)
  when ([500]) if (!this.hasRetried): do {
    retry(req);
    this.hasRetried = true;
```

```
  }
  default: throwSomething();
}
```

The intention of whether or not something is assigned is confusing. Likely status is not meant to be used in the match statement, but it could be by the current rules. **This hides programmer intention**, largely for aesthetic reasons.

Also, note for the following:

```
match (res) {
  when ({ status: 200, body, ...rest }):
    handleData(body, rest)
  when ({ status, destination: url })
      if (300 <= status && status < 400):
    handleRedirect(url)
  when ({ status: 500 }) if (!this.hasRetried): do {
    retry(req);
    this.hasRetried = true;
  }
  default: throwSomething();
}
```

If body or destination is undefined, we will go to “throwSomething()”, which may not be intentional in some cases. In addition -- rest is purely assignment.

Alternative:

```
function maybeRetry(res) {
  // for more complex cases, we can use functions which return
  // a boolean.
  return res.status == 500 && !this.hasRetried;
}

match (res) {
  let { status, body, ...rest } when { status: 200, body }:
    handleData(body, rest)
  let { destination: url }
        when { status and status >= 300 and status < 400, destination
}:
```

```
    handleRedirect(url)
  when maybeRetry.bind(this): { // can also be a higher order fn
      retry(req);
      this.hasRetried = true;
  }
  default: throwSomething();
}

match (command) {
  let [_, dir] when ['go', dir and ('north' or 'east' or 'south' or
'west')]: go(dir);
  let [_, item] when ['take', (/[a-z]+ ball/ and { weight })]:
take(item);
  default: lookAround();
}
```

The two match statements in the alternative have the same structure, as opposed to the currently proposed syntax. The search for the assigned variable is easier, as its position is fixed to the left like any assignment. In addition, we can be more intentional here. You will likely have noticed that status is unnecessary:

```
match (res) {
  let { body, ...rest } when { status: 200, body }:
    handleData(body, rest)
  let { destination: url }
        when { status and status >= 300 and status < 400, destination
}:
    handleRedirect(url)
  when maybeRetry.bind(this): { // can also be a higher order fn
      retry(req);
      this.hasRetried = true;
  }
  default: throwSomething();
}

match (command) {
  let [_, dir] when ['go', ('north' or 'east' or 'south' or 'west')]:
go(dir);
```

```
  let [_, item] when ['take', (/[a-z]+ ball/ and { weight })]:
take(item);
  default: lookAround();
}
```

In addition, the proposal as it is now, over emphasizes the cases that you would destructure and test at the same time. The above would more likely be:

```
match (res) {
  let { body, ...rest } when { status: 200 }:
    handleData(body, rest)
  let { destination: url }
        when { status and status >= 300 and status < 400 }:
    handleRedirect(url)
  when maybeRetry.bind(this): { // can also be a higher order fn
      retry(req);
      this.hasRetried = true;
  }
  default: throwSomething();
}
```

The argument that the repetition of `let` may become frustrating is legitimate. There are cases when we want conflation. Consider

```
<Fetch url={API_URL}>
  {props => match (props) {
    when ({ loading }): <Loading /> // unused var
    when ({ error }): do {
      console.err("something bad happened");
      <Error error={error} />
    }
    when ({ data }): <Page data={data} />
  }}
</Fetch>
```

That should be possible to control intentionally. Consider this alternative:

```
<Fetch url={API_URL}>
  {props => match (props) {
```

```
    when { loading }: <Loading />
    let when { error }: do {
      console.err("something bad happened");
      <Error error={error} />
    }
    when ({ data }): <Page data={data} />
  }}
</Fetch>
```