Pattern Matching - JS Object (Dictionary) Matching
A Prototyped Implementation

Alan Litteneker
CS237 - Prototyping Programming Languages

Prerequisites: Before getting into the design itself, there are some necessary and relevant things to define.

- Before designing this expansion, I added support for Javascript regular expressions and capture groups. These kinds of regex patterns are used in many of the following examples.

```
match('hello world', when(/hello world/), function(x) { return x })
        →       'hello world'
match('hello world', /(\w+) (\w+)/, function(x) { return x })
        →       ['hello', 'world']
```

- I also added a non-binding wildcard, which I called _nc.

```
match([ 'hello', 'world' ], [ _nc, _ ], function(x) { return x })
        →       'world'
```

- The rest of the examples here use the following objects:

```
var person = {
        name: 'John Smith', address: '55 Main St.',
        email: 'jsmith@smithy.net', tel1: '8475938045',
        tel2: '3198378866', tel3: '6297653562'
};
var othPerson = { name: 'John Smith', company_email_addr: 'john@smithy.net' };
```

Requirements: This design was motivated by for several different additional match capabilities.

- Complex (non-literal) patterns for both keys and values.
- Perform pattern-level binding processing with values and keys.
- Ensure that the matched object has the desired key-value pairs, and lacks any undesired pairs.
- Aggregate many key/values together to allow for easier processing in the called function.

Simple Cases: Given that we're trying to match patterns with objects, it would be nice if we could give an object as a pattern definition, and we can simply do just that. The value of the match is an object with the key-values containing the data bound when matching each, eg. the keys for which the value pattern produced a binding. If no data is bound for the entire object, the original object is passed. For example:

```
match(person, {name: _, email: 'jsmith@smithy.net'}, function(x) { return x });
        →       {name: 'John Smith'}
match(othPerson,
        { name: 'John Smith', company_email_addr: /\w+@smithy.net/ },
                function(x) { return x }
);
        →       { name: 'John Smith', email: 'john@smithy.net' }
match(person,
        { name: /(\w+)\s(\w+)/, email: /(\w+)@smithy.net/ },
                function(x) { return x }
);
        →       { name: ['John', 'Smith'], email: ['john'] }
```

Extra Keys: Behind the scenes, the object pattern is being converted to a more complex pattern type with a function called matchObj. It accepts a couple of other parameters, one of which is a boolean flag as to whether or not to accept an object which contains extra key-value pairs (eg. kv's unspecified in pattern). For example:

```
match(othPerson, matchObj({ name: _ }, false), function(x) { return x });
        →       Error: Match Failure
```

```
match(othPerson,
        matchObj({ name: _ }, false),
                function(x) { return x },
        matchObj({ name: 'John Smith', company_email_addr: _ }, false),
                function(x) { return x }
);
        →       { email: 'john@smithy.net' }
```

Key Patterns: We've seen how to use relatively complex *value* patterns, but we haven't used any non-literal *key* patterns. And here we come to a problem with our current pattern style: Javascript/JSON requires that all object keys be either strings or numbers. So, we need another object pattern syntax, and the matchObj function provides one. It's a little difficult to explain, so here's a simple example.

Let's say we want to get an email address out of our object, but we are not sure which key which key it corresponds to. We can find it with the following:

```
match(othPerson,
        matchObj([{ name: 'email', key_pattern: _nc, val_pattern: when(/^\w+@\w+\.\w+$/) }]),
                function(x) { return x }
)
        →       { email: 'john@smithy.net' }
```

Remember when I mentioned that the matchObj converted the simple JSON object to a more complex pattern? This syntax style explicitly defines that complex pattern. It consists of an array of objects, each of which is a single key-value pattern definition, where the order in which the matches are attempted is the order they appear in the array. This definition needs a couple of things, including an arbitrarily complex pattern for both the key and value ('key_pattern' and 'val_pattern'). It also needs a 'name' value, which will be used as the key for whatever bindings the patterns for this definition produce.

This quickly allows us to do some pretty cool stuff. Let's say we want to get several phone numbers out of our object. We can fetch them all, and aggregate them together under a single key with the following pattern.

```
match(person,
        matchObj([{ name: 'phone', key_pattern: /tel\d+/, val_pattern: when(/\d+/) }]),
                function(x) { return x }
)
        →       { phone: [ '8475938045', '3198378866', '6297653562' ] }
```

But what if we also want the key pattern to produce some bindings? Well, we can do that too. It results in the value for the key being an object with a 'keys' and 'values' array, the corresponding elements of which share indices. For example:

```
match(person,
        matchObj([{ name: 'phone', key_pattern: _, val_pattern: when(/\d+/) }]),
                function(x) { return x }
)
        →       { phone: {
                        keys:   [ 'tel1',       'tel2',       'tel3'       ],
                        values: [ '8475938045', '3198378866', '6297653562' ]
                } }
```

One final note on this syntax. Two definitions which share the same name will be aggregated to the same binding key in the returned object. However, the requirement for one to be matched is not fulfilled by the other being matched (if required; see below for details). If two key definitions are required and share the same name, both must be required for the object to be a match.

More Parameters: This second syntax style opens the floodgates for extra specifications. There are two additional parameters worth mentioning which can be set in a key-value definition.

- The first is the 'type' parameter which can be set to 'required,' 'allowed,' or 'disallowed.' The default is 'required,' as a programmer will usually want the specified key-value pairs to actually appear in the bound object.
    - Required means that if none of the key-value pairs in the object match this key-value definition, the object is not a match for this pattern.
    - Allowed means that there may or may not be a match for this key-value definition for a matched object. This allows for binding processing and aggregation on optional keys.
    - Disallowed means that no key-value pair that matches this key-value definition may exist in a matched object. This allows for exclusionary key settings.
- The second is the 'multiple' parameter. This boolean flag controls whether or not a given key-value definition may match multiple key-value pairs in the object (eg. with the phone number example above). By default, it is set to false for patterns defined via a JSON object, and true for those defined with the array syntax.