Java Scripting

Elixir tools use JavaScript as the embedded scripting language. The implementation is based on the Mozilla Rhino engine (http://www.mozilla.org/rhino/).

Here are the links to common JavaScript issues:

- Addition and Concatenation
- Variables with Spaces

JavaScript can be embedded in literal text by using variable substitution syntax where the first character is an equals "=" sign. For example,

```
Today is ${=new java.util.Date()}
```

will embed today's date into the field. The alternative is to declare the field with a script source:

```
"Today is " + new java.util.Date();
```

Both variants are equivalent in terms of performance and result. However, the ${=...} syntax can be used anywhere a variable substitution is allowed, for example in a URL:

```
http://www.localhost.com/error-${=getYMD()}.log
```

where getYMD is a JavaScript function you have written to return a string of the form 20050804 thus forming a URL *http://www.localhost.com/error-20050804.log*.

JavaScript is a weakly typed language where every function allows any number of parameters to be passed. For example, if you have a multiply function:

```
function multiply(a,b)

{

return a * b;

}
```

then, you can call this many ways.

The obvious way would be to pass in two numbers:

```
multiply(4,5);
```

The multiply function returns 20.0

However, you can also pass in two strings:

```
multiply("4","5");
```

and it still returns 20.0

Of course, if you pass in:

```
multiply("hello","5");
```

then the function =returns NaN (not a number)=, but still does not fail. This is all due to weak typing - Javascript does not define or care about the types of parameters.

Further, JavaScript does not care about the number of parameters.

```
multiply(4,5,6);
```

It returns 20.0 because the last parameter is ignored.

```
multiply(4);
```

It returns NaN= because only one parameter is supplied. None of these are considered errors.

It is possible to write the multiply function as follows:

```
function multiply()

{

var result = 1;

for (i=0;i<arguments.length;i++)

result = result * arguments[i];

return result;

}
```

and call it with:

```
multiply(4); // returns 4.0

multiply(4,5); // returns 20.0

multiply(4,5,6); // returns 120.0
```

Thus the number of parameters is also flexible. The function does not even need to declare that it accepts any parameters. As JavaScript supports all of these variants, any validation check can only check whether the syntax is valid, and not whether it meets your specific semantic requirements. The syntax defines whether it is valid JavaScript.

Here's an excerpt from a web site, easily found with google, searching for syntax and semantics:

> "A language is a set of valid sentences. What makes a sentence valid? You can break validity down into two things: syntax and semantics. The term syntax refers to grammatical structure whereas the term semantics refers to the meaning of the vocabulary symbols arranged with that structure. Grammatical (syntactically valid) does not imply sensible (semantically valid), however. For example, the grammatical sentence "cows flow supremely" is grammatically ok (subject verb adverb) in English, but makes no sense."

Typical syntax errors in JavaScript would be:

```
int i; // no need to declare types

var for; // variable name can't be a keyword

for (i=0) // for loop needs three clauses

multiply(4 5); // missing comma

multiply(multiply(4,5),multiply(2,3); // missing closing )
```

All of these are caught by the Validate check because they are all syntax errors. The JavaScript Validate function is checking syntax; however, due to the nature of the language, it is not possible to automatically check JavaScript semantics.