Smålang

A **small**, structurally typed, embeddable **lang**uage for JSON-to-JSON transformation.

> Named after Småland, the kids' play area at Ikea.

Key ideas

JSON is Smål

Smålang is a superset of JSON. Strings, numbers, booleans, null, objects and arrays work in a familiar way.

```
{ "hello": "world" }
```

This is a Smål program returns `"world"` when invoked with the argument `"hello"` and `void` when invoked with any other argument.

Types are values

Smålang values may be *abstract* or *concrete*. An abstract value is very similar to a "type" — it defines the set of concrete values that "match" it.

Unlike types, Smålang abstract values are first-class: they can be bound to names, passed to and returned from functions, etc.

```
{ "x": int, "y": int }
# "int" is an abstract value that matches all concrete integers
```

Objects are functions

All values in Smålang can be called, like functions; what that does varies.

When JSON objects are called with string keys as arguments, they return the corresponding values. Smålang's function syntax is a generalization of this: keys can be anything, including abstract values.

```
fibonacci <- {
    0: 1,
    1: 1,
    int -> n: fibonacci(n - 1) + (fibonacci(n - 2))
    # The key here is "int"; we'll discuss the "-> n" soon.
}
```

Functional

Every value is immutable. There is no way to, e.g. update one property of an object; you just construct a new one.

Every function is pure: outputs depend only on inputs. `now` is fixed for the duration if the computation. There is no `random` or IO.

First-class functions. They can be bound to names, passed to and returned from functions, etc.

Lexical closures. Inner functions can use bindings from outer functions.

Syntax

Smålang has:

- Everything in JSON: `"`, `{` … `}`, `[` … `]`, `:`, `,`
- The `->` and `<-` operators for binding values to names
- *Postfix functions*, declared with `{:` instead of `{`
- Function calls: `sqrt 9` (or `9 sqrt` for postfix functions)
- Parentheses for specifying order-of-operations
- Semicolons to end declarations
- The rest/spread operator `...`

So it *doesn't* have:

- Keywords. `true`, `null`, `int` etc. are values from the standard library.
- Any other operators. `+`, `=`, `<=` are functions from the standard library.
- Conditionals (`if`, `switch`, `match`) and loops (`for`, `while`).
- Special operator precedence.
- Exception handling (`try` … `catch`)
- Nominal typing, inheritance, classes, interfaces, traits, …

Anatomy a Smål function

Functions are made up of `,`-separated *cases*. Each case consists of:

- A match / bind expression terminated by `:`
- Zero or more *declarations* terminated by `;`
- A value expression

Match / Bind

We've already seen binding: it uses the `<-` and `->` operators and work symmetrically in both directions.

```
greeting <- "Hello"

"World" -> planet
```

Match / Bind

In functions, we *match* and bind at the same time:

```
{ int -> n : ... }, # Match any integer and bind it to the name "n"
```

You can “de-structure” and match on parts too.

```
{ { "age": int -> age, "name": string -> name, ...any } : ... }
```

Declarations

These help break down computations into separate steps, and bind intermediate values to names. An example:

```
# This function accepts an object with "Price" and "Qty" properties
# and returns the total with shipping.
{
    { "Price": num -> price, "Qty": num -> qty }:          # match & bind
        total <- price * qty;                             # declaration
        shipping <- total < 100 {: true: 10, false: 0 }; # declaration
        total + shipping                                  # return value
}
```

Expressions

Expressions are sequences of values and are evaluated left-to-right, with no operator precedence.

```
1 + 1 * 2   # Unlike most other languages, this is 4.
1 + (1 * 2) # This is 3.
```

Given a sequence `foo bar baz`, Smålang will evaluate it as `(foo bar) baz`.

Normally, `foo bar` evaluates `foo` with the argument `bar`. However if `bar` is a *postfix function*, it evaluates `bar` with the argument `foo` instead.

Postfix functions

Here's factorial implemented as a normal function

```
factorial <- { 1: 1, num -> n: factorial (n - 1) * n }
```

and as a postfix function named `!`

```
! <- {: 1: 1, num -> n: (n - 1)! * n }
```

Special values

- `void` represents the absence of a value (like JS `undefined`).
- numbers, booleans, `null` and `void` always return `void` when called
- strings concatenate when called: `"hello" 3` returns `"hello3"`
- arrays are basically objects with integer keys.

&, | and refinement

- Recap: abstract values are *sets* of concrete values
- When `foo` *matches* `bar`, it means `foo` is a subset of `bar`
- `|` and `&` operators perform the union and intersection operations
- `&` with a boolean expression using bound values performs *refinement*

```
{
  int -> n & n > 0 : ... # Do something with n, a positive integer
}
```

Tricks

Infix operators

Here's how the `+` operator is implemented in Smål. We use function currying with a mix of postfix and prefix functions so we can write `1 + 1`.

```
+ <- {: num -> a : { num -> b :
    # Native code to add a and b
} }
```

Iteration

The basic idea is to have functions like filter, map and reduce. Details TBD.

Todo

- Refine the syntax and remove rough edges
- Add an effect system for doing IO, exceptions, futures