Axolotl

> Hubo un tiempo en que yo pensaba mucho en los axolotl. Iba a verlos al acuario del Jardín des Plantes y me quedaba horas mirándolos, observando su inmovilidad, sus oscuros movimientos. Ahora soy un axolotl. *Julio Cortázar*

Vision / Purpose

Axo is how we imagine Haskell as a Lisp. While Haskell has a "clean syntax" it is full syntactic sugar that makes it harder to reason/learn about, also, this makes source-code transformations not accesible to most programmers. Haskell is already good at meta-programming, with type classes, template haskell, etc. But it lacks some of the power that macros can offer. Therefore, we propose a new programming language with Haskell semantics, but with Lisp syntax, plus some syntax features that we think are convenient for programming haskell style.

Main objective, category

A haskell-like Lisp. Strongly typed purely functional programming language.

Language requirements

Basic Elements

All the language is case sensitive

Tokens:

name	Description	examples
Integer	one or more digits [0-9]+	`1`, `234`, `3556`
Float	one or more digits, a decimal point followed by one or more digits	`1.23`, `0.5`, `2234`
string literal	any char or escaped char between double quotes	`"a string"`, `""`
char literal	only one char or eschaped char between single quotes	`'a'`,`'\n'`
varId	any sequence of non reserved chars, starting with a lowercase char	`a`, `solve`, `x0`,`>>=`
typeId	any sequence of non reserved chars, starting with a uppercase char	`List`, `Maybe`, `Int`

Syntax Diagrams

Grammar:

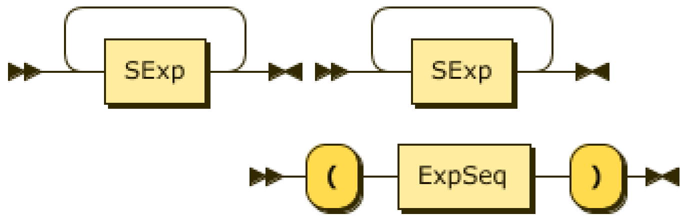

Figure 1: SExp

Figure 2: ExpSeq

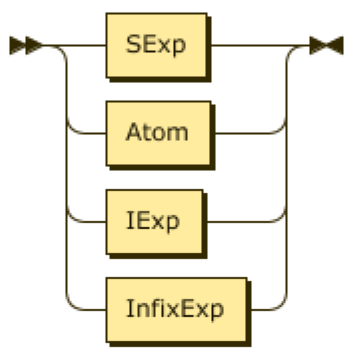

Figure 3: Exp

Lexems:

Semantic Characteristics

Our language is divided into two fields: values and types. Values are data whereas types are sets of values.

Values and expressions

- The language has semantics close to Haskell.
- It has a strong static type system, therefore every expression has a type.
- All variables are immutable.

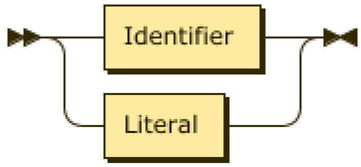

Figure 4: Atom

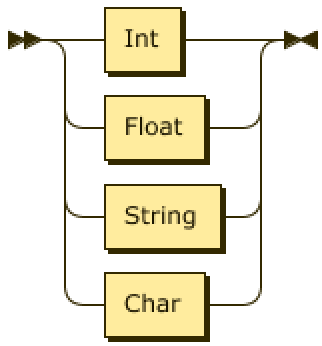

Figure 5: Literal

Figure 6: IExp

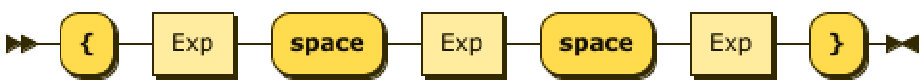

Figure 7: InfixExp

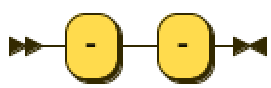

Figure 8: CommentSingleLine

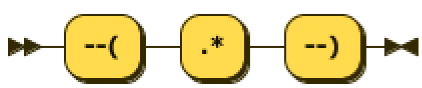

Figure 9: CommentMultiLine

Figure 10: digit

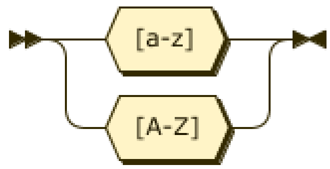

Figure 11: letter

Figure 12: decimal

Figure 13: Int

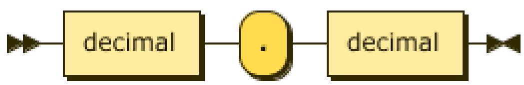

Figure 14: Float

Figure 15: UniChar

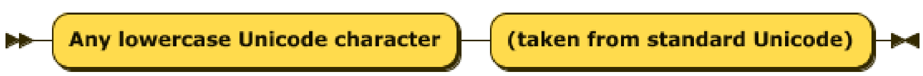

Figure 16: LowerUni

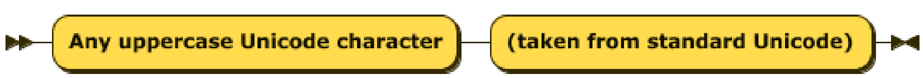

Figure 17: UpperUni

Figure 18: Char

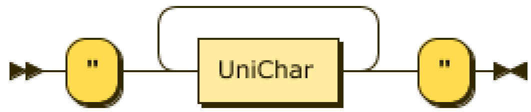

Figure 19: String

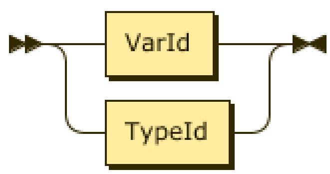

Figure 20: Identifier

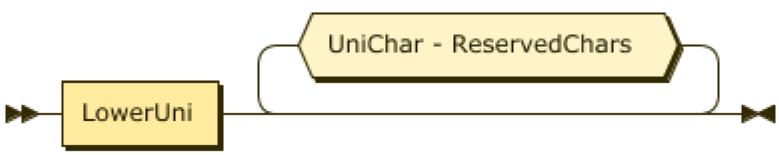

Figure 21: VarId

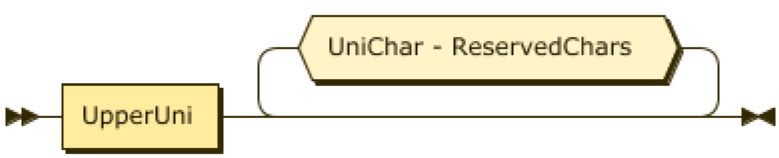

Figure 22: TypeId

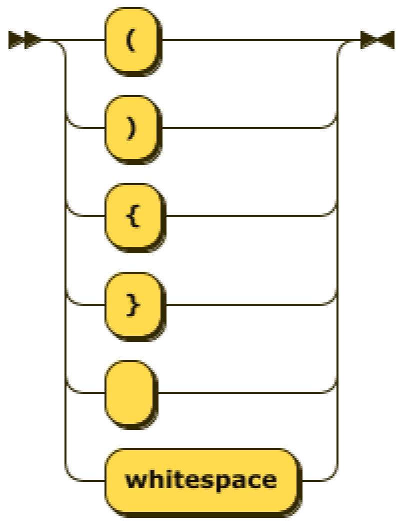

Figure 23: ReservedChars

Figure 24: ValidSymbol

Figure 25: NotDoubleQuote

- Functions are automatically curried, therefore it's easy to partially apply, the disadvantage is that there are no variable arity functions.

The field of values include primitives and user defined. Primitives like a number, a character or a function, and user defined like a tree. These are all first class. Expressions are a combination of values by the means of function application. For example `(+ 2 3)` or the function `( x -> (* x x))`. There is a very special value: undefined. When undefined is evaluated, the program crashes. Undefined allows us to define partial functions. Similarly there is a very special function: error. This function receives a String s and returns undefined, which will crash the program with the error message s.

It is important to note that Axo does not differentiate between functions and operators, because the simplicity of the syntax allows an identifier to be composed of only symbols.

1. Looping

   To write something that can be executed multiple times, one should write recursive function:

```
define (loop x)
    if {x == 0}
    0
    (loop {x - 1})
```

```
define (fibonnaci n)
    cond ({n == 0} 0)
    ({n == 1} 0)
    (else (fibonnaci {n - 1}))
```

2. On Folds

   From a functional programming perspective, folds are called catamorphisms, this is important because they are equivalent to a **for-each loop** in other languages. Therefore, if we can add folds to our language, we can express this loops.

   fold right associative:

```
define (foldr f end xs)
    if (null? xs)
    end
    (f (head xs) (foldr f end (tail xs)))
```

fold left associative:

```
define (foldl f a xs)
    if (null? xs)
    a
    (foldl f (f a (head xs)) (tail xs))
```

Special Functions and Forms

1. Input/Output

    1. IO primitives

Name	Description
`putChar`	writes a char
`putStr`	writes a string
`putStrLn`	writes a string with a newline at the end
`getChar`	reads one char
`getLine`	reads a complete line
`getContents`	reads all the content

    2. IO higher level

Name	Description
`write`	writes data in a way that can be read by the machine
`read`	reads input and returns the data parsed
`display`	prints data in a way that can be read by a human
`displayLn`	prints data and a newline at the end, in a way that can be read by a human

2. Math Functions

    1. Integers

Name	Description
`+`	integer sum
`-`	integer substraction
`*`	integer product
`/`	integer division
`mod`	modulo

    2. Floats

Name	Description
`+.`	float sum
`-.`	float substraction
`*.`	float product
`/.`	float division
`sqrt`	square root function

Name	Description
`log`	logarithm of x
`exp`	exponential of x

3. Special Forms

Name	Description	Grammar
`if`	evals predicate, and evals only one of the expressions depending on the result	(if <predicate> <if-true> <if-false>)
`cond`	evaluates the clauses one by one, in the first clause that succeeds, the corresponding expression is evaluated and returned.	(cond (<clause1> … <clausen>)) where clausex = (<predicatex> <expressionx>
`data`	a data type definition	(data <typeName> <type expression>)
`type`	type alias	(type <typeName> <type expression>)
`and`	short-circuit `and` (also known as conditional and)	(and <expression1> <expression2>)
`or`	short-circuit `or` (also known as conditional or)	(or <expression1> <expression2>)

Name	Description	Grammar
`lambda`	a lambda abstraction (can also be written with the unicode	(lambda (<arguments>) <body>)
`let`	local bindings	(let <var name> <expression>)
`define`	top level definition of a function or variable	(define <var name> <expression>) or (define (<function name> <args>) <expression>)

Extensions

Name
`defmacro`
`class`
`instance`

Data Types

1. Type System

   The field of types include type values and type variables. Type values are monomorphic where as type variables are polymorphic. Neither of these are first class. A type value, or just called "type", can be understood as a set of possible values. Type variables can be understood as a set of any type. We can view type variables as generics in other languages. Type values include Int or Int -> Int. Polymorphic types include the function head which type is List a -> a. Therefore this functions is defined forAll a types.

   The primitive types are: Integer, Float, Character.

2. On Types

   A sum type is the union different constructors for the same type, for example:

   ```
   (data Bool {True | False})
   ```

   On the contrary, product types can be understood as a tuple of any two types (their cartesian product), the types can be different, for example:

   ```
   (data Point (Pt Int Int))
   ```

   Product Types are like having some "type arguments" to a data constructor, while sum types are different constructors.

   An example of combining both of these types:

   ```
   (data (Node a)) (data Tree {(Node (Tree a) (Tree a)) | (Leaf a)})
   ```

   In this case, the Tree can be either a Tree with two branches, or an empty Tree. This case is also a good example of a Recursive Type.

   Written in infix notation:

   ```
   (data Tree {{(Tree a) Node (Tree a)} | (Leaf a)})
   ```

Language and OS used for development

Axo is written in Haskell, and developed on MacOS and Debian linux.

More Axo code

Append function in Axo:

```
(define (append Nil ys) ys) :: {List a -> List a -> List a}
(define (append {x : xs} ys)
{x : (append ys xs)})

(define (++ Nil ys)

(infixr ++)
(define ++ append)
```

Concat method in Axo:

```
define (concatHello str)
{"hello" ++ str}

define main
```

```
let (contents (readLn))
(write (concatHello contents))
```

Bibliography

https://www.haskell.org https://www.haskell.org/tutorial/goodies.html
https://docs.racket-lang.org/hackett/index.html http://tunes.org/overview.html
https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours/Towards_a_Standard_Library
http://dev.stephendiehl.com/fun/006_hindley_milner.html#types

Features we would like to have

Type Classes

A possible extension to the type system are type classes, which are a constraint over a polymorphic type, that forces a type to be an instance of that class. This means that it implements a specific associated function. We can think of classes as interfaces in other languages. Examples include: Num, Show, Read, Ord and Eq.

Meta-programming

eval time compilation time **development time**

We think that compilers, programming languages and tools are not always designed with ergonomics in mind. There is a special focus on formality, yet as an example error reporting is ad-hoc. The users of programming languages (the programmers) are given text-focused tools only to develop, mantain and refactor code. There is no intrinsic reason this should be the case. Our main objective is, to provide meta-programming tools to the programmer.

Why is meta-programming feared? Our hypothesis is that its unpredicability makes it unfit for a program while it's running, and to a lesser extent, during compilation (just ask a programmer if they use macros in their own programs). Now there are exceptions to this phenomenon such as hygienic macros in lisps, or ruby object system.