Op language (unfinished yet)

What is Op ?

Op is a language built around operators. You can think of operators as functions written in infix notation. This project is intended to teach lexing, parsing, interpreting, and compiling concepts, not to create a production-ready language. Currently, Op is an interpreted language implemented in OCaml.

I welcome any feedback or ideas. You can reach me at:

- Email: akskwyx@gmail.com
- Discord: .skwyx

Objectives (features)

- Create a usable language
- Allow custom operator precedence
- Support defining operators with infix notation using regex patterns (see Examples)
- *(Optional)* Add or remove parameters to change an operator's arity (concept stage)

Examples

Refer to the code for detailed behavior (documentation in progress)

Fibonacci function

```
fibo n -> n <= 1 ? 1 : fibo >> n-1 + fibo >> n-2
```

Iter function

```
// Declare the iter operator
iter <- i <- i + 1

// Initialize variable k to 0
k <- 0

// While k <= 5, apply iter to k
iter >> k ** k <= 5
```

Regex-based operator example

```
{ (\[a\] \+\* \[b\])+ } <- ((
  + !< 2
  * !< 3
  a.i + b.i * a.(i+1) + b.(i+1) ** i < (a.len - 1)
```

```
))

var <- 1 +* 2 +* 3 +* 4
$var
```

1. Declare a regex operator matching `(value +* value)+`.
2. Set `+` precedence to 2 (default 3) and `*` to 3 (default 2), so `+` binds tighter.
3. Sum each pair of `a` and `b`, multiply all results, and assign to `var`.
4. Print `var`: `1 + 2 * 3 + 4 = 3 * 7 = 21` (remember the precedence changes).

TODO

- **Project**
  - Add documentation
- **Parsing**
  - Change list type to enable O(1) append operations
  - Add panic mode to the parser to handle multiple errors
  - Optimize the parser to avoid unnecessary backtracking
  - Modify some point on the grammar for it to be more user-friendly
- **Interpreting**
  - Optimize the interpreter :
    - Better handling of recursive calls
    - Add right tail recursion
    - Better handling of environments
  - Implement error handling