Op language (Ong)

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 learn 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)

Basic documentation

Operators :

- `<-` : Declaration operator
- `>>` : Application operator
- `?  :` : Ternary operator (if-then-else)
- `**` : Loop operator (while)
- `$` : Print operator
- `{ }` : Block
- `||` : Or operator
- `&&` : And operator
- `=` : Equality operator
- `!=` : Inequality operator
- `<`, `<=`, `>`, `>=` : Comparison operators
- `+`, `-`, `*`, `/` : Arithmetic operators
- `!`, `-` : Unary operators (logical not, negation)
- `!<` : Precedence change operator

Precedence table :

```
0 :
1 : declaration (<-)
2 : statement (S)
3 : expression (E)
```

```
4 : logic_or (||)
5 : logic_and (&&)
6 : equality (=)
7 : comparison (>)
8 : term (+)
9 : factor (*)
10 : unary (!)
11 : call (>>)
12 : primary (P)
```

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

```
iter i <- i + 1;;
k <- 0;
{$k k <- iter >> k;} ** k <= 5
```

Result :

```
012345
```

List implementation

```
create <- null;
isEmpty l <- l = null;
add l v <- isEmpty >> l ? {value <- v; next <- null;} : {value <- v; next <- l;};
remove l <- isEmpty >> l ? $"Trying to remove from an empty list" : l >> next;
print l <- {m <- l; {$(m >> value) $" - " m <- remove >> m;} ** !isEmpty >> m};

l <- create;
$isEmpty >> l
$" "
l <- add >> l 0;
l <- add >> l 1;
print >> l;
```

Result :

```
false 1 - 0 -
```

Precedence change

```
$(1 + 2 * 2)
$" "
+ !< 9
* !< 8
$(1 + 2 * 2)
```

Result :

```
5 6
```

How to use

Execute your code with the command :

```
./bin/main.bc <your_file.op>
```

You can move the `main.bc` file where you want and is the only file you need to run the interpreter.

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
- **Features**
  - Enable changing operator precedence for specific operators and not only for groups of operators
  - Add custom operator