1 Installation

In order to compile you need:

- `ocaml`
- `ocamlbuild`
- `make`

You can simply type `make` in the root directory to build everything, or you can specify which part you want to build with

- `make bin` to build the compiler
- `make test` to compile the examples
- `make doc` to compile this document
- `make archive` to build tar and zip source archives.

2 The Language

This is a minimalist functional language with strict left to right evaluation. Here is the grammar of the language. The terminals are:

- Identifiers (`IDENT`) are sequences of character matching the regular expression `[A-Za-z][A-Za-z0-9]*`.
- Integers (`INT`) are sequences of character matching the regular expression `-?[0-9]+`.
- Strings (`STRING`) are sequences of character enclosed by double quotes.
- Booleans are `true` and `false`

A program is an expression (`EXPR`) composed of:

- constants: `INT`, booleans and `STRING`.
- variables : `IDENT`.
- functions: **fun** $\texttt{IDENT}_1$ ... $\texttt{IDENT}_n$ `->` `EXPR`. For example:

```
fun x y z -> y
```

- applications: **(** `EXPR`$_0$ `EXPR`$_1$ ... `EXPR`$_n$ **)**. It applies the arguments $(\texttt{EXPR}_i)_{i>0}$ to `EXPR`$_0$. **Do not forget the brackets** or you'll be surprised!

- let-in: **let** `IDENT` = `EXPR`$_1$ **in** `EXPR`$_2$. It computes expression `EXPR`$_1$, assigns its value to variable `IDENT` and computes `EXPR`$_2$.

- sequences: **(** `EXPR`$_1$ **;** ... **;** `EXPR`$_n$ **)**. It computes sequentialy the expressions `EXPR`$_1$ to `EXPR`$_n$. The last one is the value of the expression. **Do not forget the brackets** or you'll be surprised!

- if-then-else: **if** `EXPR`$_1$ **then** `EXPR`$_2$ **else** `EXPR`$_3$. If `EXPR`$_1$ evaluates to `true`, then `EXPR`$_2$ is evaluated otherwise `EXPR`$_3$ is. The value of the expression is the one of the branch being evaluated.

3 Primitives

Primitives are functions written directly in TEXor in shell.
Primitives written in TEXmust have an argumet pattern of the form `#1...#n`. They must return a value by redefining the macro `acc`. Finally, they must be placed in the file `tex/extern.tex`. For example, the addition over integers is defined in `tex/extern.tex` by:

```
\def\wcplus#1#2%
{{%
\count255=#1%
\advance\count255 by #2%
\xdef\acc{\the\count255}%
}}%
```

Primitives written in shell must return a value by redefining the variable `ACC`. They must be placed in the file `shell/extern.sh`. For example, the addition over integers is defined in `shell/extern.sh` by:

```
wcplus () {
  ACC=$(($1 + $2))
}
```

Primitives can be used as identifiers. But they **must** be declared first. A primitive is declared by the statement: **extern** *primitive-name arity* **;;** at the begining if the program. For example, `wcplus` is declared by:

```
extern wcplus 2 ;;
```

4 Compiling and Executing

A program is compiled to TEXby:

```
wc.native -t -i input-file -o output-file
```

and to POSIX shell by:

```
wc.native -s -i input-file -o output-file
```

`output-file` can be run as any other shell script or TEXfile. For example, the program `first.wc`:

```
extern myprint    1 ;;
extern newline    1 ;;
extern printstats 1 ;;

let phrase = fun qui ->
 let action = fun quoi ou -> ( (myprint "The great " ) ;
                               (myprint qui          ) ;
                               (myprint " eats "     ) ;
                               (myprint quoi         ) ;
                               (myprint " in the "   ) ;
                               (myprint ou           ) ;
                               (newline 0            )
                             )
 in let pommes = action "apples"
    and poires = action "pears"
    in ( (pommes "garden." ) ;
         (pommes "garage." ) ;
         (poires "castle.") ;
         (poires "linvngroom."  )
       )
in ( ( phrase "Pierre" ) ;
     ( phrase "Paul"   ) ;
     ( phrase "Jaques" )
   )
```

can be compied and ran to TEXand shell by

```
wc.native -t -i first.tex -o first.tex
wc.native -s -i first.tex -o first.sh
```

Its output is:

```
The great Pierre eats apples in the garden.
The great Pierre eats apples in the garage.
The great Pierre eats pears in the castle.
The great Pierre eats pears in the linvngroom.
The great Paul eats apples in the garden.
The great Paul eats apples in the garage.
The great Paul eats pears in the castle.
The great Paul eats pears in the linvngroom.
The great Jaques eats apples in the garden.
The great Jaques eats apples in the garage.
The great Jaques eats pears in the castle.
The great Jaques eats pears in the linvngroom.
```