B5 - Compilation

B-GCC-500

Koak

Kind Of Advanced Kaleidoscope

1.0

{EPITECH.}

Koak

group size:	3-4
repository name:	koak
repository rights:	ramassage-tek
language:	Kotlin, Scala, Haskell, F#, OCaml
compilation:	stack, opam & Makefile, sbt, gradle, MSBuild

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

For those of you who'd chose F#, .NetCore works really well, and ,combined with Jet-Brains Rider, allows you to work on Linux or Windows in the exact same way.

You need to implement a compiler/interpreter of the **KOAK - Kind Of Alternative Kaleidoscope** language as defined in this document.
You have to use:

- a functional language,
- PEG parsing techniques,
- LLVM to produce a binary (or JIT "Just In Time" evaluation).

The KOAK language is very close to the original Kaleidoscope Language.

The **B-FUN-500** module is a strongly advised requirement.

All the librairies/functions that handle parsing are forbidden.

+ 1 - GRAMMAR

In your preferred language, you must implement a PEG parser for the KOAK language and build an AST.

You must handle syntax errors gracefully.

Don't forget to modify the grammar to handle spaces (simple space, '\t', '\n', '\r').

+ 2 - REPL - READ-EVAL-PRINT-LOOP

When invoked without parameter, your **koak** executable enters a Read-Eval-Print-Loop.

Terminal

```
~/B-GCC-500> ./koak
KOAK, compiler/interpreter
ready> def test(x : double):double 1.0 + 2.0 + x;
define double @test(double %x) {
entry:
     %addtmp = fadd double 2.000000e+00, 1.000000e+00
     %addtmp1 = fadd double %addtmp, %x
     ret double %addtmp1
}
6
```

The information line (before the ready>) is implementation dependant.

Use the JIT functionalities of LLVM to evaluate the expressions.

Dumping the LLVM equivalent is **mandatory**.

You are linked to the **libc**, so "extern" definition are just a way to access the libc functions in KOAK.

+ 3 - COMPILER

When you invoke your **koak** executable with one parameter, you enter compilation mode.
Your program must then produce an executable binary (ELF/ 64 bit):

Terminal

```
~/B-GCC-500> ls
helloworld.koak
~/B-GCC-500> koak helloworld.koak
~/B-GCC-500> ls
a.out helloworld.koak
```

All your **top_expr** are collected in definition order and constitute your **main** entry point.
The ELF produced is dynamically linked to libc (check it with *ldd*).

+ 4 - ADD SIMPLE TYPE INFERENCE

Implement a hindley-milner type system.

You can find an example of implementation here.

If you implement KOAK in Scala or Haskell, you surely found a basic implementation of this algorithm.
However, you need to really understand it because you need to adapt it to the KOAK.

Hindley milner is used in functional languages, but as you see, KOAK is an imperative language.
Don't panic, it's OK.

In fact, HM builds a tree from AST that represents the relation between types (known and/or unknown) and try to validate it.

You could infer the type of your variables so that in **prototype_args** the sequence `: type` becomes optional.
That would be nice.

+ 5 - ADD CHAR AND STR

Add the 'char' and 'str' types.
To do so, you need to add this in the **literal** rules:

```
string_const <- '"' (~('"' / '\' . ))* '"'

char_const <- "'" (~("'" / '\' . )) "'"
```

Add the support of the '[]' operator as a **postfix** rule.

```
postfix <- primary (paren_expr / index_expr)*

index_expr <- '[' expression ']'
```

In **index_expr**, **expression** is evaluated as **int** value and provides the index of the relative **left associative primary** of type **str**.
The result type is **char**.

For the **extern** definition, the **str** type is translated to `char*`.

+ 6 - CUSTOM FEATURES

Fun and creativity mode ON!

You could add all specific features in your KOAK compiler.
But:

- don't override **mandatory** features
- avoid implementing **ad-hoc polymorphism**. HM Type system doesn't deal with it.

KOAK GRAMMAR

Here is the complete grammar of the language.
Enjoy it.

```
stmt <- kdefs* #eof

kdefs <- ext_def
        / local_def
        / top_expr

ext_def <- "extern" defs ';'
local_def <- "def" defs ';'
top_expr <- expressions ';'

defs <- prototype expressions

prototype <- (
            'unary' . decimal_const?
            / 'binary' . decimal_const?
            / identifier
            )
            prototype_args

prototype_args <- '(' (identifier ':' type)* ')' ':' type

type <- 'int' / 'double' / 'void'

while_expr <- 'while' expression 'do' expressions

for_expr <- 'for' identifier '=' expression
        ',' identifier '<' expression
        ',' expression 'in' expressions

if_expr <- 'if' expression "then" expressions ("else" expressions)?



expressions <- for_expr
            / if_expr
            / while_expr
            / expression (':' expression)*

expression <- unary (#binop (#left_assoc unary / #right_assoc expression))*

unary <- #unop unary / postfix

postfix <- primary call_expr?

primary <- identifier
        / literal
        / paren_expr

call_expr <- '(' (expression (',' expression)*)? ')'

paren_expr <- '(' expressions ')
```

```
identifier <- [a-zA-Z][a-zA-Z0-9]*

dot <- '.' !'.'

exp <- ('e' / 'E') ('+'/'-')? [0-9]+

decimal_const <- [0-9]+

hexadecimal_prefix <- '0' ('x'/'X')

hexadecimal_digit <- [0-9a-fA-F]

hexadecimal_const <- hexadecimal_prefix hexadecimal_digit+

octal_digit <- [0-7]

octal_const <- '0' octal_digit+

double_const <- (decimal_const dot [0-9]* / dot [0-9]+ ) exp?

literal <- hexadecimal_const
        / octal_const
        / decimal_const
        / double_const
```

+ SPECIAL RULES

#binop is a special rule that allows you (thanks to PEG) to handle classical binary operator or user defined binary operators.

#left_assoc is just a function/rule saying 'true' if the previously read **#binop** is left associative, 'false' otherwise.

#right_assoc is just a function/rule saying 'true' if the previously read.

#binop is right associative, 'false' otherwise.
Here is a listing of all binary operator that you need to implement.
The **assoc** column, provides the associativity of the operator.

GROUP	RULE	PRIORITY	ASSOC	GROUP	RULE	PRIORITY	ASSOC
assignement	'=' ! '='	10	right	**comparison**	("==" / "!=")	70	left
	"+="	10	right		"<="	80	left
	"-="	10	right		">="	80	left
	"*="	10	right		'<' ! '<'	80	left
	"/="	10	right		'>' ! '>'	80	left
	"%="	10	right				
	"<<="	10	right	**calculus**	'+' ! ('+' / '=')	100	left
	">>="	10	right		'-' ! ('-' / '=' / '>')	100	left
	"&="	10	right		'*'	110	left
	"^="	10	right		'/'	110	left
	"|="	10	right		'%'	110	left
logical	"||"	20	left				
	"&&"	30	left				
binary	"|" ! ("|" / "=")	40	left				
	"^" ! "="	50	left				
	"&" ! ("&" / "=")	60	left				
	"<<" ! "="	90	left				
	">>" ! "="	90	left				

#unop is a special rule that allows you to handle classical unary operators or user defined unary operators.
Classical unary operator are:

- **!** for logical not.
- **~** for binary inversion.
- **+** for unary plus.
- **-** for unary minus.

+ SEMANTIC

- **decimal_const**, **octal_const**, **hexadecimal_const** rules are typed as 'int'.
- **double_const** is typed as 'double'.
- In **primary**, **identifier** is typed relatively to is nearest definition.
  If not previously defined, its type is the generic **TypeVar** (see HM Type system).
- In **postfix**, if **primary** is typed as a function, **call_expr** matches the function prototype.
- **expressions** is typed as:
  - In the **expression** branch, we use the type of the **last expression**.
  - In **for_expr** branch, we use the type of the **for_expr** rule.
  - In **if_expr** branch, we use the type of the **if_expr** rule.
  - In **while_expr** branch, we use the type of the **while_expr** rule.
- **for_expr** has the same type as its **expressions**.
- **while_expr** has the same type as its **expressions**.
- **if_expr** has the same type as the **expressions** of the **then** branch. If you have an **else**, it has the same type as the **then** branch.
- **local_def** creates a function in the global scope. If already defined and we are in compiler mode, an error will be reported by the compiler. In interpreter mode (REPL), the function is just redefined.