Student Name: Your name, your.email@ntut.edu.tw
Student ID: Your student ID number,
Instructor: Jyun-Ao Lin, jalin@ntut.edu.tw

The goal is to build a compiler for a tiny fragment of the Koka language[1], called Petit Koka in the following, to x86-64 assembly. This is a fragment 100% compatible with Koka, in the sense that any `Petit Koka` program is also a correct Koka program. This will allow the latter to be used as a reference. This topic describes precisely `Petit Koka`, as well as the nature of the work requested.

1 Syntax

We use the following notations in grammars

$\langle rule \rangle^*$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero)
$\langle rule \rangle_t^*$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero), with separator t
$\langle rule \rangle^+$	repeats $\langle rule \rangle$at least once
$\langle rule \rangle_t^+$	repeats $\langle rule \rangle$ at least once, with separator t
$\langle rule \rangle?$	use $\langle rule \rangle$ optionally
$(\langle rule \rangle)$	grouping

Be careful not to confuse "$*$" and "$+$" with "`*`" and "`+`" that are Python symbols. Similarly, do not confuse grammar parentheses with terminal symbols `(` and `)`.

1.1 Lexical Conventions

Spaces and tabs are considered blanks. Comments can take two forms:

- starting with `/*` and extending to `*/`, and cannot be nested;
- starting with `//` and extending to the end of the line.

Identifiers follow the regular expression $\langle ident \rangle$:

$$
\begin{array}{lcl}
\langle digit \rangle & ::= & \texttt{0-9} \\
\langle lower \rangle & ::= & \texttt{a-z} \mid \_ \\
\langle upper \rangle & ::= & \texttt{A-Z} \\
\langle other \rangle & ::= & \langle lower \rangle \mid \langle upper \rangle \mid \langle digit \rangle \mid \texttt{-} \\
\langle ident \rangle & ::= & \langle lower \rangle \ \langle other \rangle^* \ \texttt{'}^*
\end{array}
$$

The following identifiers are keywords:

[1] https://koka-lang.github.io/

```
elif    else  fn   fun  if
return  then  val  car
```

Integer literals follow the regular expression $\langle integer \rangle$:

$$\langle integer \rangle \quad ::= \quad \texttt{-?}\ (\texttt{0} \mid \texttt{1-9}\ \langle digit \rangle^*)$$

String literals are written between quotes (`"`). There are four escape sequences: `\"` for the character ", `\\`for the character \, `\t` for the tab character and `\n` for a carriage return. A string cannot contain a carriage return.

Significant indentation The Koka language uses alignment in the source text to define the structure of blocks. More precisely, token `(beginning of block),` (end of block) and `;` (separation of elements of a block) are sometimes added implicitly by lexical analysis. Syntactic analysis is then performed in the traditional way (see next section). Here is an example, with the source text on the left and the sequence of tokens produced by the lexical analyzer on the right:

source text	produced tokens

`fun main()`
`  repeat(10)`
`    println("a")`
`    println("b")`
`  println("c")`
	`;`
	`fun main() {`
	`    repeat(10) {`
	`        println("a");`
	`        println("b"); } ;`
	`    println("c"); } ;`

Explanation:

- The first carriage return, at the end of the first empty line, produced a token `;`
  (and we can check that the grammar accommodates semicolons at the beginning of the file).
- The second carriage return produced a token `{` because the token `repeat` is on a larger column than the token `fun`. The same goes for the third carriage return.
- The fourth carriage return (after `println("a")`) produced a token `;` because the indentation is identical to that of the previous line.
- The fifth carriage return produced on the one hand the tokens `;` and `}`, because the indentation decreases, then the token `;` because it finds the indentation of `repeat`.
- Finally, the end of the file has a similar effect of finding an indentation on column 0, which first produces the tokens `;` and `}` then the token `;`, as in the previous point.

Definitions We introduce two classes of tokens for the purposes of the algorithm below. A token is said to be *an end of continuation* if it is part of this list of tokens:

```
+ - * / % ++ < <= > >= == != && || ( { ,
```

Similarly, a token is said to be *a start of a continuation* if it is part of this list of tokens:

```
+ - * / % ++ < <= > >= == != && || then else elif ) } , -> { = . :=
```

These two notions are used to determine whether an action should be performed when a carriage return is encountered.

Algorithm We use a *stack* that contains integers. These are the columns of the currently active indentation blocks. These integers are strictly increasing, with the largest being at the top of the stack. Initially, the stack contains the integer 0.

We read the tokens one by one, with a tool of our choice, and we permanently retain the last token that was produced (denoted `last` in the following). When we encounter a carriage return, we read the next token `next`, we denote its column c and we perform the following actions:

- if $c > m$ where m is the top of the stack, then
  1. if `last` is not an end of continuation and `next` is not a start of continuation, then emit `{` ;
  2. if the last token emitted is `{`, then push c ;
  3. emit `next`.
- if $c \leq m$, where m is the top of the stack, then
  1. as long as $c < m$
     (a) pop m and give m the new value at the top of the stack,
     (b) emit `{` if `next` $\neq$ `}`
  2. if $c > m$, fail;
  3. otherwise, ie, $c = m$, and
     (a) if `last` is not an end of continuation and `next` is not a start of continuation, then emit `;`,
     (b) emit `next`.

To handle the end of file correctly, we can consider that the token `EOF` is on column 0 and treat it in the general case (second point above). Finally, any emission of the token `}` is preceded by the emission of a token `;`.

Suggestion Start by writing a traditional lexical analyzer (with a tool of the type `lex`), then build on top of it a second lexical analyzer that inserts the additional tokens on the fly. For example, we could use a queue to store the tokens waiting to be sent to the parser, in order to find the type expected from a lexical analyzer that sends the tokens one by one. Many tests, negative and positive, are provided to help understand this mechanism and to fine-tune the lexical analyzer

1.2 Syntax

The grammar of source files is given in Fig. 1. The entry point is the non-terminal $\langle file \rangle$. Associativity and priorities are given below, from lowest to strongest priority

operation	**associativity**	**priority**
`if`	–	lowest
`||`	left	
`&&`	left	
`:=, ==, !=, >, >=, <, <=`	–	↓
`+, -, ++`	left	
`*, /, %`	left	
`~, !`	–	
`.`, {}(postfix), `fn`(postfix)	–	strongest

Block In a block (non-terminal $\langle block \rangle$), the last element must be an expression ($\langle expr \rangle$) and not a `val` or `var` declaration.

Syntax sugar Some constructions exist only in syntax

syntax	translation
e `.` x	$x(e)$
$e(e_1, \ldots, e_n)$ `fn` f	$e(e_1, \ldots, e_n)$, `fn` f
e `fn` f	e(`fn` f) if e is not a callee
$e(e_1, \ldots, e_n)$ { e }	$e(e_1, \ldots, e_n$, `fn () {`b`}`)
e { e }	e(`fn () {`b`}`) if e is not a callee
...`elif`...	...`else if`...
`if` e_1 `then` e_2	`if` e_1 `then` e_2 `else` {}
`if` e_1 `return` e_2	`if` e_1 `return` e_2 `else` {}

$$
\begin{array}{lcl}
\langle file \rangle & ::= & ;^* \ (\langle decl \rangle^* \ :^+)^* \ \texttt{EOF} \\
\langle decl \rangle & ::= & \texttt{fun} \ \langle ident \rangle \ \langle funbody \rangle \\
\langle funbody \rangle & ::= & \texttt{(} \ \langle param \rangle^*_{,} \ \texttt{)} \ \langle annot \rangle? \ \langle expr \rangle \\
\langle param \rangle & ::= & \langle ident \rangle \ \texttt{:} \ \langle type \rangle \\
\langle annot \rangle & ::= & \texttt{:} \ \langle result \rangle \\
\langle result \rangle & ::= & (\texttt{<} \langle ident \rangle^*_{,} \texttt{>})? \ \langle type \rangle \\
\langle type \rangle & ::= & \langle atype \rangle \\
& | & \langle atype \rangle \ \texttt{->} \ \langle result \rangle \\
& | & \texttt{(} \ \langle type \rangle^*_{,} \ \texttt{)} \ \texttt{->} \ \langle result \rangle \\
\langle atype \rangle & ::= & \langle ident \rangle \ (\texttt{<} \langle type \rangle \texttt{>})? \\
& | & \texttt{(} \langle type \rangle \texttt{)} \\
& | & \texttt{( )} \\
\langle atom \rangle & ::= & \texttt{True} \mid \texttt{False} \mid \langle integer \rangle \mid \langle string \rangle \mid \texttt{( )} \\
& | & \langle ident \rangle \\
& | & \texttt{(} \langle expr \rangle \texttt{)} \\
& | & \langle atom \rangle \ \texttt{(} \langle expr \rangle^*_{,} \texttt{)} \\
& | & \langle atom \rangle \ \texttt{.} \ \langle ident \rangle \\
& | & \langle atom \rangle \ \texttt{fn} \ \langle funbody \rangle \\
& | & \langle atom \rangle \ \langle block \rangle \\
& | & \texttt{[} \ \langle expr \rangle^*_{,} \ \texttt{]} \\
\langle expr \rangle & ::= & \langle block \rangle \\
& | & \langle bexpr \rangle \\
\langle bexpr \rangle & ::= & \langle atom \rangle \\
& | & \texttt{~} \langle bexpr \rangle \\
& | & \texttt{!} \langle bexpr \rangle \\
& | & \langle bexpr \rangle \ \langle binop \rangle \ \langle bexpr \rangle \\
& | & \langle ident \rangle \ \texttt{:=} \ \langle bexpr \rangle \\
& | & \texttt{if} \ \langle bexpr \rangle \ \texttt{then} \ \langle expr \rangle \ (\texttt{elif} \ \langle bexpr \rangle \ \texttt{then} \ \langle expr \rangle)^* \ (\texttt{else} \ \langle expr \rangle)? \\
& | & \texttt{if} \ \langle bexpr \rangle \ \texttt{return} \ \langle expr \rangle \\
& | & \texttt{fn} \ \langle funbody \rangle \\
& | & \texttt{return} \ \langle expr \rangle \\
\langle block \rangle & ::= & \{ \ ;^* \ (\langle stmt \rangle \ ;^+)^* \ \} \\
\langle stmt \rangle & ::= & \langle bexpr \rangle \\
& | & \texttt{val} \ \langle ident \rangle \ \texttt{=} \ \langle expr \rangle \\
& | & \texttt{var} \ \langle ident \rangle \ \texttt{:=} \ \langle expr \rangle \\
\langle binop \rangle & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{++} \mid \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

Figure 1: Grammar of Petit Koka

2 Static Typing

Once the syntactic analysis is successfully performed, the conformity of the source file is checked. The types are of the following form:

$$\begin{array}{rcl} \tau & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \texttt{list}\langle\tau\rangle \mid \texttt{maybe}\langle\tau\rangle \mid (\tau, \ldots, \tau) \to \kappa \\ \kappa & ::= & \epsilon/\tau \\ \epsilon & \subseteq & \{\ \texttt{div, console}\ \} \end{array}$$

A type τ represents a value type and a type κ represents a computation type, which combines a value type and an effect ϵ. The latter indicates a possible non-termination (`div`, for divergence) and/or a possible writing to the standard output (`console`).

Environment of type The local typing environment, denoted by Γ, contains an ordered sequence of variable declarations of the form $\texttt{val}\ x : \tau$ or $\texttt{var}\ x : \tau$. We denote by $\Gamma + \texttt{val}\ x : \tau$ the environment Γ extended with a new variable declaration (same thing with `var`). A previous declaration of x in Γ is, if applicable, replaced by this new declaration.

2.1 Type of expression

We introduce the judgment $\Gamma \vdash e : \kappa$ meaning "in the context Γ, the expression e is well-typed of type κ". This judgment is defined by the following inference rules

$$\frac{c \text{ constant of type } \tau}{\Gamma \vdash c : \emptyset/\tau} \qquad \frac{\texttt{val}\ x : \tau \in \Gamma}{\Gamma \vdash x : \emptyset/\tau} \qquad \frac{\texttt{var}\ x : \tau \in \Gamma}{\Gamma \vdash x : \emptyset/\tau} \qquad \frac{\texttt{var}\ x : \tau \in \Gamma \qquad \Gamma \vdash e : \epsilon/\tau}{\Gamma \vdash x := e : \emptyset/\tau}$$

$$\frac{\Gamma \vdash e : \epsilon/\texttt{int}}{\Gamma \vdash \texttt{\~{}}e : \epsilon/\texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{int} \qquad \Gamma \vdash e_2 : \epsilon_2/\texttt{int} \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1\ op\ e_2 : \epsilon_1 \cup \epsilon_2/\texttt{int}}$$

$$\frac{\Gamma \vdash e : \epsilon/\texttt{bool}}{\Gamma \vdash \texttt{!}e : \epsilon/\texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{int} \qquad \Gamma \vdash e_2 : \epsilon_2/\texttt{int} \qquad op \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \epsilon_1 \cup \epsilon_2/\texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{bool} \qquad \Gamma \vdash e_2 : \epsilon_2/\texttt{bool} \qquad op \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\ op\ e_2 : \epsilon_1 \cup \epsilon_2/\texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{string} \qquad \Gamma \vdash e_2 : \epsilon_2/\texttt{string}}{\Gamma \vdash e_1\ \texttt{++}\ e_2 : \epsilon_1 \cup \epsilon_2/\texttt{string}}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{list}\langle\tau\rangle \qquad \Gamma \vdash e_2 : \epsilon_2/\texttt{list}\langle\tau\rangle}{\Gamma \vdash e_1\ \texttt{++}\ e_2 : \epsilon_1 \cup \epsilon_2/\texttt{list}\langle\tau\rangle}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{bool} \qquad \Gamma \vdash e_2 : \epsilon_2/\tau \qquad \Gamma \vdash e_3 : \epsilon_3/\tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \epsilon_1 \cup \epsilon_2 \cup \epsilon_3/\tau}$$

$$\frac{\forall i,\ \Gamma \vdash e_i : \epsilon_i/\tau}{\Gamma \vdash \texttt{[}e_1, \ldots, e_n\texttt{]} : \bigcup \epsilon_i/\texttt{list}\langle\tau\rangle}$$

$$\frac{}{\Gamma \vdash \{\ \} : \emptyset/\texttt{unit}} \qquad \frac{\Gamma \vdash e : \kappa}{\Gamma \vdash \{e\} : \kappa} \qquad \frac{\Gamma \vdash e_1 : \epsilon_1/\tau_1 \qquad \Gamma \vdash \{e_2;\ \ldots\} : \epsilon_2/\tau_2}{\Gamma \vdash \{e_1; e_2; \ldots\} : \epsilon_1 \cup \epsilon_2/\tau_2}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\tau_1 \qquad \Gamma + \texttt{val}\ \ x : \tau_1 \vdash \{e_2; \ldots\} : \epsilon_2/\tau_2}{\Gamma \vdash \{\texttt{val}\ \ x = e_1; e_2; \ldots\} : \epsilon_1 \cup \epsilon_2/\tau_2}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\tau_1 \qquad \Gamma + \texttt{var}\ \ x : \tau_1 \vdash \{e_2; \ldots\} : \epsilon_2/\tau_2}{\Gamma \vdash \{\texttt{var}\ \ x = e_1; e_2; \ldots\} : \epsilon_1 \cup \epsilon_2/\tau_2}$$

$$\frac{\Gamma + \texttt{val}\ \ x_1 : \tau_1 + \cdots + \texttt{val}\ \ x_n : \tau_n \vdash e : \kappa}{\Gamma \vdash \texttt{fn}\ \ (x_1 : \tau_1, \ldots, x_n : \tau_n)\ e : \emptyset/(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \kappa}$$

$$\frac{\Gamma + \texttt{val}\ \ x_1 : \tau_1 + \cdots + \texttt{val}\ \ x_n : \tau_n \vdash e : \kappa}{\Gamma \vdash \texttt{fn}\ \ (x_1 : \tau_1, \ldots, x_n : \tau_n) : \kappa\ e : \emptyset/(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \kappa}$$

$$\frac{\Gamma \vdash e : \epsilon/((\tau_1, \ldots, \tau_n)) \to \epsilon_f/\tau \qquad \forall i,\ \Gamma \vdash e_i : \epsilon_i/\tau_i}{\Gamma \vdash e(e_1, \ldots, e_n) : \epsilon \cup \epsilon_f \cup \bigcup \epsilon_i/\tau} \qquad \frac{\Gamma \vdash e_1 : \epsilon_1/\tau_1}{\Gamma \vdash \texttt{return}\ \ e_1 : \epsilon_1/\tau}$$

Typing of predefined functions

$$\frac{\Gamma \vdash e : \epsilon/\tau \qquad \tau \in \{\texttt{unit}, \texttt{bool}, \texttt{int}, \texttt{string}\}}{\Gamma \vdash \texttt{println}(e) : \epsilon \cup \{\texttt{console}\}/\texttt{unit}} \qquad \frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{maybe}\langle\tau\rangle \qquad \Gamma \vdash e_2 : \epsilon_2/\tau}{\Gamma \vdash \texttt{default}(e_1, e_2) : \epsilon_1 \cup \epsilon_2/\tau}$$

$$\frac{\Gamma \vdash e : \epsilon/\texttt{list}\langle\tau\rangle}{\Gamma \vdash \texttt{head}(e) : \epsilon/\texttt{maybe}\langle\tau\rangle} \qquad \frac{\Gamma \vdash e : \epsilon/\texttt{list}\langle\tau\rangle}{\Gamma \vdash \texttt{tail}(e) : \epsilon/\texttt{list}\langle\tau\rangle}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{int} \qquad \Gamma \vdash e_2 : \epsilon_2/\texttt{int} \qquad \Gamma \vdash e_3 : \epsilon_3/((\texttt{int}) \to \epsilon/\texttt{unit})}{\Gamma \vdash \texttt{for}(e_1, e_2, e_3) : \epsilon_1 \cup \epsilon_2 \cup \epsilon_3 \cup \epsilon/\texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/\texttt{int} \qquad \Gamma \dashrightarrow e_2 : \epsilon_2/(() \to \epsilon/\texttt{unit})}{\Gamma \vdash \texttt{repeat}(e_1, e_2) : \epsilon_1 \cup \epsilon_2\epsilon/\texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \epsilon_1/(() \to \epsilon_3/\texttt{bool}) \qquad \Gamma \vdash e_2 : \epsilon_2/(() \to \epsilon_4/\texttt{unit})}{\Gamma \vdash \texttt{while}(e_1, e_2) : \epsilon_1 \cup \epsilon_2 \cup \epsilon_3 \cup \epsilon_4 \cup \{\texttt{div}\}/\texttt{unit}}$$

2.2 Type of a file

Declarations in a file are processed in the order in which they appear. Functions are added to the environment as they are parsed. A function can only refer to previous functions (or to itself). Each function can only be defined once.

Function declaration A function f is introduced by a declaration of the following form

$$\texttt{fun}\ \ f(x_1 : \tau_1, \ldots, x_n : \tau_n)\ (:\kappa)?\ e$$

with an optional return type κ and a function body that is an expression e. The parameters x_i must have distinct names. We check that the body e is well typed in the environment Γ of the previous functions extended with f and its parameters, that is,

$$\Gamma + \texttt{val } f : ((\tau_1, \ldots, \tau_n) \to \kappa) + \texttt{val } x_1 : \tau_1 + \cdots + \texttt{val } x_n : \tau_n \vdash e : \kappa$$

where κ is the type given by the user, if any, or otherwise a minimal effect calculation type (in the sense of inclusion). If the body e of the function refers to the function f (in a recursive call or even by passing f to another function), then κ must include the effect `div`. Finally, it is worth checking that any expression `return` e_1 appearing in the body of the function actually returns a value of the type expected by κ.

Function `main` The source file must contain a `main` function without parameters. Its return type is free.

2.3 Limitations compared to Koka

The `Petit Koka` language suffers from a number of limitations compared to Koka. In particular,

- `Petit Koka` has fewer keywords than Koka;
- Koka allows not to indicate the type of a function parameter (and infers it).

However, your compiler will never be tested on programs that are incorrect in the sense of `Petit Koka` but correct in the sense of Koka

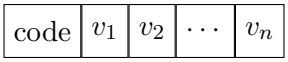
2.4 Remarks

It is strongly recommended to proceed construction by construction, by systematically compiling and testing your project at each stage. Many tests are provided on the Teams, with a script to launch your compiler on these tests. If in doubt about a point of semantics, you can use the Koka compiler as a reference. You can also draw inspiration from its error messages for your compiler. In the next phase (code production), certain information from typing will be necessary. You are advised to anticipate these needs by programming typing functions that do not simply browse the abstract syntax trees resulting from syntactic analysis but return new ones, including types. These new abstract syntax trees may differ more or less from the trees resulting from syntactic analysis.

3 Code Generation

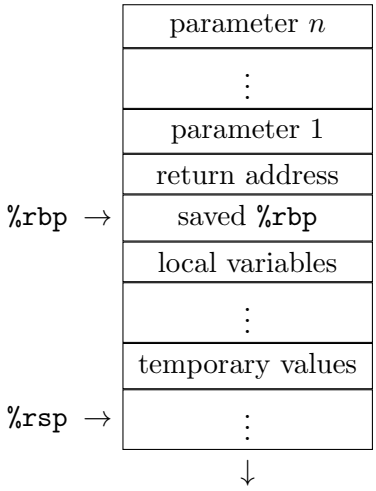
The aim is to produce a simple but correct compiler. In particular, we do not attempt to do any kind of register allocation, but simply use the stack to store any intermediate calculations. Of course, it is possible, and even desirable, to use some x86-64 registers locally. We will not try to free the memory. We propose here a compilation scheme of indicative title. You are free to make any other choice.

Value Representation Integers are immediately represented as signed 64-bit integers (although Koka's `int` type is an arbitrary-precision integer type). The value `()` is represented by the integer 0. Booleans are represented by integers, with the same convention as x86-64: 0 represents `False` and 1 represents `True`. A string is an address to a heap-allocated block containing a string terminated by a `0` character (as in C). A list is either the integer 0 (the empty list) or an address to a two-word heap-allocated block containing the value at the head of the list and the rest of the list. A value of type `maybe`$\langle\tau\rangle$ is represented either by the integer 0 (nothing) or by a pointer to a one-word heap-allocated block containing a value of type τ. Finally, a function type value is represented by a closure, that is, an address to a block of $n+1$ words allocated on the heap of the form



where the first word contains a code pointer and the following words contain the variables $v_1, \ldots, v_n$ captured in the closure.

Compilation scheme Each function in `Petit Koka` is compiled to an assembler function that takes all its arguments on the stack and places its return value in the `%rax` register. If the function is a closure, it is passed as the very first parameter. It makes things easier to consider any function as a closure, including global functions



It is strongly recommended to proceed in two steps, starting with an explanation of closures. That being said, it is still possible to start with the fragment without any anonymous function, in a simpler compilation scheme and to add anonymous functions only in a second step. However, be careful with the `for`/`repeat`/`while` constructions which are higher-order functions in Koka. You must then either treat them in a special way, or wait until you have added the processing of closures.

We can write some functions directly in assembler (string and list concatenation, `println` instances, etc.) and add this to the code produced by the compiler. Be careful, before calling C library functions such as `printf` or `malloc`, it is important to align the stack, i.e. to ensure that `%rsp` is a multiple of 16 before calling. The easiest way to do this is to define small assembler functions that call the library functions after correctly aligning the stack, for example like this:

```
my_malloc:
```

```
        pushq   %rbp
        movq    %rsp, %rbp
        andq    $-16, %rsp      # 16-byte stack alignment
        movq    24(%rbp), %rdi  # pass malloc argument into the stack
        call    malloc
        movq    %rbp, %rsp
        popq    %rbp
        ret
```

If we take this approach, however, it is not necessary to align the stack for its own functions.

4 Project Assignment

The project must be done **alone or in pair**. It must be delivered on Teams, as a compressed archive containing a directory with your student ID(s). Inside this directory, source files of the compiler must be provided (no need to include compiled files). The command `make` must create the compiler, named `kokac`. The command `make clean` must delete all the files that make has produced and leave only the source files in the directory. Of course, the project can be compiled with tools other than make (for example dune if the project is made in OCaml) and the `Makefile` is then reduced to a few lines only to call these tools.

The archive must also contain **a short report** explaining the technical choices and, if any, the issues with the project and the list of whatever is not delivered. The report can be in format ASCII, Markdown, or PDF.

The command line of `kokac` accepts an option (among `--parse-only` and `--type-only`) and exactly one file with extension `.koka`. It must then perform the syntactic analysis of the file. In the event of a lexical or syntactic error, this must be reported as precisely as possible, by its nature and location in the source file. The following format will be adopted for this reporting:

```
File "test.koka", line 4, characters 5-6:
syntax error
```

If the file is parsed successfully, the compiler must terminate with code 0 if option `--parse-only` is on the command line. Otherwise, the compiler moves to static type checking. Any type error must be reported as follows:

```
File "test.koka", line 4, characters 5-6:
this expression has type int but is expected to have type string
```

The location indicates the filename name, the line number, and the column number. Feel free to design your own error messages. The exit code must be 1.

If the file is type-checked successfully, the compiler must exit with code 0 if option `--type-only` is on the command line. In case of an error by the compiler itself, the compiler must terminate with exit code 2. The –type-only option indicates to stop the compilation after the semantic analysis (typing). It therefore has no effect in this first part, but it must be honored nonetheless.

Otherwise, the compiler generates x86-64 assembly code in file `file.s` (same name as the input file, but with extension `.s` instead of extension `.koka`). The x86-64 file will be compiled and run as follows

```
gcc file.s -o file
./file
```

possibly with option `-no-pie` on the `gcc` command line. The result displayed on standard output should be identical to that given by the following command:

```
koka --console=raw -v0 -e file.koka
```

An Extension of Your Choice Last, but not least, you have to implement an extension of your choice. This can be

- the support of another Python construct;
- more static analysis;
- awesome error messages;
- a compiler optimization;
- etc.

This extension must be described in the report and illustrated with test files.