projet de compilation

Petit Koka

version 1 — 24 octobre 2024

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Koka[1], appelé Petit Koka par la suite, produisant du code x86-64. Il s'agit d'un fragment 100% compatible avec Koka, au sens où tout programme de Petit Koka est aussi un programme Koka correct. Ceci permettra notamment d'utiliser ce dernier comme référence. Le présent sujet décrit précisément Petit Koka, ainsi que la nature du travail demandé.

1 Syntaxe

Dans la suite, nous utilisons les notations suivantes dans les grammaires :

$\langle règle \rangle^\star$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle règle \rangle^\star_t$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle règle \rangle^+$	répétition de la règle $\langle règle \rangle$ au moins une fois
$\langle règle \rangle^+_t$	répétition de la règle $\langle règle \rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle règle \rangle?$	utilisation optionnelle de la règle $\langle règle \rangle$ (*i.e.* 0 ou 1 fois)
($\langle règle \rangle$)	parenthésage ; attention à ne pas confondre ces parenthèses avec les terminaux `(` et `)`

1.1 Conventions lexicales

Espaces et tabulations constituent des blancs. Les commentaires peuvent prendre deux formes :

- débutant par `/*` et s'étendant jusqu'à `*/`, et ne pouvant pas être imbriqués ;
- débutant par `//` et s'étendant jusqu'à la fin de la ligne.

Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

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

Par ailleurs, un tiret (`-`) dans un identificateur doit être précédé d'une lettre ou d'un chiffre et suivi d'une lettre s'il n'est pas en dernière position.

Les identificateurs suivants sont des mots clés :

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

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

Les constantes entières obéissent à l'expression régulière ⟨*integer*⟩ suivante :

⟨*integer*⟩ ::= -? (0 | 1–9 ⟨*digit*⟩*)

Une chaîne de caractères ⟨*string*⟩ s'écrit entre guillemets ("). Il y a quatre séquences d'échappement : \" pour le caractère ", \\ pour le caractère \, \t pour le caractère de tabulation et \n pour un retour chariot. Une chaîne ne peut pas contenir de retour chariot.

1.2 Indentation significative

Le langage Koka utilise l'alignement dans le texte source pour définir la structure des blocs. Plus précisément, des lexèmes { (début de bloc), } (fin de bloc) et ; (séparation des éléments d'un bloc) sont parfois ajoutés implicitement par l'analyse lexicale. L'analyse syntaxique est ensuite réalisée de manière traditionnelle (voir section suivante). Voici un exemple, avec à gauche le texte source et à droite la suite de lexèmes qui est produite par l'analyseur lexical :

texte source	lexèmes produits

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

Explications :

- Le premier retour chariot, à la fin de la première ligne qui est vide, a produit un lexème ; (et on pourra vérifier que la grammaire s'accommode de points-virgules en début de fichier).
- Le deuxième retour chariot a produit un lexème { car le lexème `repeat` est sur une colonne plus grande que le lexème `fun`. De même pour le troisième retour chariot.
- Le quatrième retour chariot (après `println("a")`) a produit un lexème ; car l'indentation est identique à celle de la ligne précédente.
- Le cinquième retour chariot a produit d'une part les lexèmes ; et }, car l'indentation diminue, puis le lexème ; car elle retrouve l'indentation de `repeat`.
- Enfin, la fin de fichier a un effet similaire à retrouver une indentation sur la colonne 0, ce qui produit d'abord les lexèmes ; et } puis le lexème ;, comme au point précédent.

Définitions. On introduit deux classes de lexèmes pour les besoins de l'algorithme ci-dessous. On dit qu'un lexème est *une fin de continuation* s'il fait partie de cette liste de lexèmes :

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

De même, on dit qu'un lexème est *un début de continuation* s'il fait partie de cette liste de lexèmes :

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

Ces deux notions sont utilisées pour déterminer si une action doit être effectuée lorsqu'on rencontre un retour chariot.

Algorithme. On utilise une *pile* qui contient des entiers. Ce sont les colonnes des blocs d'indentation actuellement en cours. Ces entiers sont strictement croissants, le plus grand étant au sommet de la pile. Initialement, la pile contient l'entier 0.

On lit les lexèmes un par un, avec un outil de son choix, et on retient en permanence le dernier lexème qui a été produit (noté `last` dans la suite). Lorsqu'on rencontre un retour chariot, on lit le prochain lexème `next`, on note sa colonne c et on effectue les actions suivantes :

- si $c > m$, où m est le sommet de la pile, alors
  1. si `last` n'est pas une fin de continuation et `next` n'est pas un début de continuation, alors émettre `{` ;
  2. si le dernier lexème émis est `{`, alors empiler c ;
  3. émettre `next`.
- si $c \leq m$, où m est le sommet de la pile, alors
  1. tant que $c < m$,
     (a) dépiler m et donner à m la nouvelle valeur du sommet de pile,
     (b) émettre `}` si `next` $\neq$ `}` ;
  2. si $c > m$, échouer ;
  3. sinon, c'est que $c = m$, et
     (a) si `last` n'est pas une fin de continuation et `next` n'est pas un début de continuation, alors émettre `;`,
     (b) émettre `next`.

Pour traiter la fin de fichier correctement, On peut considérer que le lexème `EOF` est sur la colonne 0 et le traiter dans le cas général (second point ci-dessus). Enfin, toute émission du lexème `}` est précédée de l'émission d'un lexème `;`.

Suggestion. Commencer par écrire un analyseur lexical traditionnel (avec un outil de type `lex`), puis construire par dessus un second analyseur lexical qui insère à la volée les lexèmes supplémentaires. On pourra par exemple utiliser une *file* pour stocker les lexèmes en attente d'être envoyés à l'analyseur syntaxique, afin de retrouver le type attendu d'un analyseur lexical qui envoie les lexèmes un par un. De nombreux tests, négatifs et positifs, sont fournis pour aider à comprendre ce mécanisme et à mettre au point l'analyseur lexical.

1.3 Syntaxe

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non terminal ⟨*file*⟩. Les associativités et précédences des diverses constructions sont données par la table suivante, de la plus faible à la plus forte précédence :

opérateur ou construction	**associativité**
`if`	—
`||`	gauche
`&&`	gauche
`==`, `!=`, `>`, `>=`, `<`, `<=`	—
`+`, `-`, `++`	gauche
`*`, `/`, `%`	gauche
`~`, `!`	—
`.`, `{}` (postfixe), `fn` (postfixe)	—

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

Blocs. Dans un bloc (non terminal $\langle block \rangle$), le dernier élément doit être une expression ($\langle expr \rangle$) et non une déclaration `val` ou `var`.

Sucre syntaxique. Certaines constructions n'existent que dans la syntaxe :

syntaxe	traduction
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` $f)$ si e n'est pas un appel
$e(e_1, \ldots, e_n)$ `{` b `}`	$e(e_1, \ldots, e_n,$ `fn () {` b `}`$)$
e `{` b `}`	$e($`fn () {` b `}`$)$ si e n'est pas un appel
`...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 `then return` e_2 `else {}`

2 Typage statique

Une fois l'analyse syntaxique effectuée avec succès, on vérifie la conformité du fichier source. Les types sont de la forme suivante :

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

Un type τ représente un type de valeur et un type κ représente un type de *calcul*, qui combine un type de valeur et un effet ε. Ce dernier indique une possible non terminaison (`div`, pour divergence) et/ou une possible écriture sur la sortie standard (`console`).

Environnement de typage. L'environnement de typage local, noté Γ, contient une suite ordonnée de déclarations de variables de la forme `val` $x : \tau$ ou `var` $x : \tau$. On note $\Gamma +$ `val` $x : \tau$ l'environnement Γ étendu avec une nouvelle déclaration de variable (même chose avec `var`). Une précédente déclaration de x dans Γ est, le cas échéant, remplacée par cette nouvelle déclaration.

2.1 Typage des expressions

On introduit le jugement $\Gamma \vdash e : \kappa$ signifiant « dans le contexte Γ, l'expression e est bien typée de type κ ». Ce jugement est défini par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \emptyset/\tau} \qquad \frac{\texttt{val}\ x : \tau \in \Gamma}{\Gamma \vdash x : \emptyset/\tau} \qquad \frac{\texttt{var}\ x : \tau \in \Gamma}{\Gamma \vdash x : \emptyset/\tau} \qquad \frac{\texttt{var}\ x : \tau \in \Gamma \quad \Gamma \vdash e : \varepsilon/\tau}{\Gamma \vdash x\ \texttt{:=}\ e : \varepsilon/\texttt{unit}}$$

$$\frac{\Gamma \vdash e : \varepsilon/\texttt{int}}{\Gamma \vdash \texttt{~}\ e : \varepsilon/\texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{int} \quad \Gamma \vdash e_2 : \varepsilon_2/\texttt{int} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1\ op\ e_2 : \varepsilon_1 \cup \varepsilon_2/\texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{int} \quad \Gamma \vdash e_2 : \varepsilon_2/\texttt{int} \quad op \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \varepsilon_1 \cup \varepsilon_2/\texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\tau \quad \Gamma \vdash e_2 : \varepsilon_2/\tau \quad \tau \in \{\texttt{bool}, \texttt{int}, \texttt{string}\} \quad op \in \{\texttt{==}, \texttt{!=}\}}{\Gamma \vdash e_1\ op\ e_2 : \varepsilon_1 \cup \varepsilon_2/\texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{bool} \quad \Gamma \vdash e_2 : \varepsilon_2/\texttt{bool} \quad op \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\ op\ e_2 : \varepsilon_1 \cup \varepsilon_2/\texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{string} \quad \Gamma \vdash e_2 : \varepsilon_2/\texttt{string}}{\Gamma \vdash e_1 \texttt{ ++ } e_2 : \varepsilon_1 \cup \varepsilon_2/\texttt{string}}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{list}\langle\tau\rangle \quad \Gamma \vdash e_2 : \varepsilon_2/\texttt{list}\langle\tau\rangle}{\Gamma \vdash e_1 \texttt{ ++ } e_2 : \varepsilon_1 \cup \varepsilon_2/\texttt{list}\langle\tau\rangle}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{bool} \quad \Gamma \vdash e_2 : \varepsilon_2/\tau \quad \Gamma \vdash e_3 : \varepsilon_3/\tau}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3/\tau}$$

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

$$\frac{}{\Gamma \vdash \texttt{\{ \}} : \emptyset/\texttt{unit}} \qquad \frac{\Gamma \vdash e : \kappa}{\Gamma \vdash \texttt{\{ } e \texttt{ \}} : \kappa} \qquad \frac{\Gamma \vdash e_1 : \varepsilon_1/\tau_1 \quad \Gamma \vdash \texttt{\{ } e_2; \ \ldots \texttt{ \}} : \varepsilon_2/\tau_2}{\Gamma \vdash \texttt{\{ } e_1; \ e_2; \ \ldots \texttt{ \}} : \varepsilon_1 \cup \varepsilon_2/\tau_2}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\tau_1 \quad \Gamma + \texttt{val } x : \tau_1 \vdash \texttt{\{ } e_2; \ \ldots \texttt{ \}} : \varepsilon_2/\tau_2}{\Gamma \vdash \texttt{\{ val } x \texttt{ = } e_1; \ e_2 \ \ldots \texttt{ \}} : \varepsilon_1 \cup \varepsilon_2/\tau_2}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\tau_1 \quad \Gamma + \texttt{var } x : \tau_1 \vdash \texttt{\{ } e_2; \ \ldots \texttt{ \}} : \varepsilon_2/\tau_2}{\Gamma \vdash \texttt{\{ var } x \texttt{ := } e_1; \ e_2 \ \ldots \texttt{ \}} : \varepsilon_1 \cup \varepsilon_2/\tau_2}$$

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

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

$$\frac{\Gamma \vdash e : \varepsilon/((\tau_1, \ldots, \tau_n)) \to \varepsilon_f/\tau \quad \forall i,\ \Gamma \vdash e_i : \varepsilon_i/\tau_i}{\Gamma \vdash e(e_1, \ldots, e_n) : \varepsilon \cup \varepsilon_f \cup \bigcup \varepsilon_i/\tau} \qquad \frac{\Gamma \vdash e_1 : \varepsilon_1/\tau_1}{\Gamma \vdash \texttt{return } e_1 : \varepsilon_1/\tau}$$

Typage des fonctions prédéfinies.

$$\frac{\Gamma \vdash e : \varepsilon/\tau \quad \tau \in \{\texttt{unit}, \texttt{bool}, \texttt{int}, \texttt{string}\}}{\Gamma \vdash \texttt{println}(e) : \varepsilon \cup \{\texttt{console}\}/\texttt{unit}} \qquad \frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{maybe}\langle\tau\rangle \quad \Gamma \vdash e_2 : \varepsilon_2/\tau}{\Gamma \vdash \texttt{default}(e_1, e_2) : \varepsilon_1 \cup \varepsilon_2/\tau}$$

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

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{int} \quad \Gamma \vdash e_2 : \varepsilon_2/\texttt{int} \quad \Gamma \vdash e_3 : \varepsilon_3/((\texttt{int}) \to \varepsilon/\texttt{unit})}{\Gamma \vdash \texttt{for}(e_1, e_2, e_3) : \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \cup \varepsilon/\texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/\texttt{int} \quad \Gamma \vdash e_2 : \varepsilon_2/(() \to \varepsilon/\texttt{unit})}{\Gamma \vdash \texttt{repeat}(e_1, e_2) : \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon/\texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \varepsilon_1/(() \to \varepsilon_3/\texttt{bool}) \quad \Gamma \vdash e_2 : \varepsilon_2/(() \to \varepsilon_4/\texttt{unit})}{\Gamma \vdash \texttt{while}(e_1, e_2) : \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \cup \varepsilon_4 \cup \{\texttt{div}\}/\texttt{unit}}$$

2.2 Typage d'un fichier

Les déclarations constituant un fichier sont traitées dans l'ordre d'apparition. Les fonctions sont ajoutées à l'environnement au fur et à mesure de leur analyse. Une fonction ne peut faire référence qu'à des fonctions antérieures (ou à elle-même). Chaque fonction ne peut être définie qu'une seule fois.

Déclaration de fonction. Une fonction f est introduite par une déclaration de la forme suivante

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

avec un type de retour κ optionnel et un corps de fonction qui est une expression e. Les paramètres x_i doivent porter des noms distincts. On vérifie que le corps e est bien typé dans l'environnement Γ des fonctions précédentes étendu avec f et ses paramètres, c'est-à-dire

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

où κ est le type donné par l'utilisateur, le cas échéant, ou sinon un type de calcul d'effet minimal (au sens de l'inclusion). Si le corps e de la fonction fait référence à la fonction f (dans un appel récursif ou même en passant f à une autre fonction), alors κ doit inclure l'effet `div`. Enfin, il convient de vérifier que toute expression `return` e_1 apparaissant dans le corps de la fonction renvoie bien une valeur du type attendu par κ.

Fonction `main`. Le fichier source doit contenir une fonction `main` sans paramètre. Son type de retour est libre.

2.3 Limitations par rapport à Koka

Le langage Petit Koka souffre d'un certain nombre de limitations par rapport à Koka. En particulier,

- Petit Koka possède moins de mots clés que Koka ;
- Koka autorise à ne pas indiquer le type d'un paramètre de fonction (et l'infère).

Cependant, votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Petit Koka mais corrects au sens de Koka.

2.4 Indications

Il est fortement conseillé de procéder construction par construction, en compilant et testant systématiquement son projet à chaque étape. De nombreux tests sont fournis sur la page du cours, avec un script pour lancer votre compilateur sur ces tests.

En cas de doute concernant un point de sémantique, vous pouvez utiliser le compilateur Koka comme référence. Vous pouvez d'ailleurs vous inspirer de ses messages d'erreur pour votre compilateur (en les traduisant ou non en français).

Dans la phase suivante (production de code), certaines informations provenant du typage seront nécessaires. Il vous est conseillé d'anticiper ces besoins en programmant des fonctions de typage qui ne se contentent pas de parcourir les arbres de syntaxe abstraite issus de l'analyse syntaxique mais en renvoient de nouveaux, contenant notamment des types. Ces nouveaux arbres de syntaxe abstraite peuvent différer plus ou moins des arbres issus de l'analyse syntaxique. Ainsi, on suggère de profiter du typage pour transformer une fonction définie par plusieurs équations en une fonction définie par une seule équation et un `case` sur l'un de ses arguments.

3 Production de code

L'objectif est de réaliser un compilateur simple mais correct. En particulier, on ne cherchera pas à faire d'allocation de registres mais on se contentera d'utiliser la pile pour stocker les éventuels calculs intermédiaires. Bien entendu, il est possible, et même souhaitable, d'utiliser localement les registres de x86-64. On ne cherchera pas à libérer la mémoire. On propose ici un schéma de compilation à titre indicatif. Vous êtes libres de faire tout autre choix.

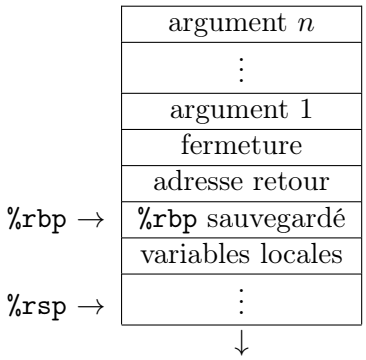
3.1 Représentation des valeurs

Les entiers sont représentés de manière immédiate par des entiers 64 bits signés (même si le type `int` de Koka est un type d'entiers de précision arbitraire). La valeur `()` est représentée par l'entier 0. Les booléens sont représentés par des entiers, avec la même convention que x86-64 : 0 représente `False` et 1 représente `True`. Une chaîne est une adresse vers un bloc alloué sur le tas contenant une chaîne terminée par un caractère `0` (comme en C). Une liste est soit l'entier 0 (la liste vide), soit une adresse vers un bloc de deux mots alloué sur le tas contenant la valeur en tête de liste et le reste de la liste. Une valeur de type `maybe`$\langle\tau\rangle$ est représentée soit par l'entier 0 (rien), soit par un pointeur vers un bloc d'un mot alloué sur le tas contenant une valeur de type τ. Enfin, une valeur de type fonction est représentée par une fermeture, c'est-à-dire une adresse vers un bloc de $n+1$ mots alloué sur le tas de la forme

code	v_1	v_2	...	v_n

où le premier mot contient un pointeur de code et les mots suivants contiennent les valeurs $v_1, \ldots, v_n$ capturées (par valeur) dans la fermeture.

3.2 Schéma de compilation



Chaque fonction de Petit Koka est compilée vers une fonction assembleur qui prend tous ses arguments sur la pile et place sa valeur de retour dans le registre `%rax`. Si la fonction est une fermeture, celle-ci est passée comme un tout premier paramètre. Cela simplifie les choses de considérer toute fonction comme une fermeture, y compris les fonctions globales.

On conseille fortement de procéder en deux temps, en commençant par une explicitation des fermetures (cf cours 8). Cela étant, il reste possible de commencer par le fragment sans aucune fonction anonyme, dans un schéma de compilation plus simple (cf cours 7) et de ne rajouter les fonctions anonymes que dans un second temps. Attention toutefois aux constructions `for`/`repeat`/`while` qui sont en Koka des fonctions d'ordre supérieur. Il faut alors soit les traiter de façon particulière, soit attendre d'avoir ajouté le traitement des fermetures.

On pourra écrire quelques fonctions directement en assembleur (concaténation de chaînes et de listes, instances de `println`, etc.) et ajouter cela au code produit par le compilateur. Attention, avant d'appeler des fonctions de bibliothèque C comme `printf` ou `malloc`, il convient d'aligner la pile, c'est-à-dire de garantir que `%rsp` est un multiple de 16 avant de faire `call`. Le plus simple pour cela consiste à définir de petites fonctions assembleur qui appellent les fonctions de bibliothèque après avoir correctement aligné la pile, par exemple comme ceci :

```
my_malloc:
        pushq   %rbp
```

```
movq    %rsp, %rbp
andq    $-16, %rsp      # alignement de la pile
movq    24(%rbp), %rdi  # argument de malloc ici passé sur la pile
call    malloc
movq    %rbp, %rsp
popq    %rbp
ret
```

Si on adopte cette approche, il n'est en revanche pas nécessaire d'aligner la pile pour ses propres fonctions.

4 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email[2], sous la forme d'une archive compressée (avec `tar` ou `zip`), appelée `vos_noms.tgz` ou `vos_noms.zip` qui doit contenir un répertoire appelé `vos_noms` (exemple : `dupont-durand`). Dans ce répertoire doivent se trouver les *sources* du compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer le compilateur, qui sera appelé `kokac`. La commande `make clean` doit effacer tous les fichiers que `make` a produits et ne laisser dans le répertoire que les fichiers sources. Bien entendu, la compilation du projet peut être réalisée avec d'autres outils que `make` (par exemple `dune` si le projet est réalisé en OCaml) et le `Makefile` se réduit alors à quelques lignes seulement pour appeler ces outils.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format texte (par exemple Markdown) ou PDF.

Partie 1 (à rendre pour le dimanche 15 décembre 18:00). Dans cette première partie du projet, le compilateur `kokac` doit accepter sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Petit Koka portant l'extension `.koka`. Il doit alors réaliser l'analyse syntaxique du fichier. En cas d'erreur lexicale ou syntaxique, celle-ci doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

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

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs[3] puisse interpréter la localisation et placer ainsi automatiquement le curseur sur l'emplacement de l'erreur. En revanche, le message d'erreur proprement dit pourra être écrit en français si vous le souhaitez. En cas d'erreur, le compilateur doit terminer avec le code de sortie 1 (`exit 1`).

Si le fichier est syntaxiquement correct, le compilateur doit terminer avec le code de sortie 0 si l'option `--parse-only` a été passée sur la ligne de commande. Sinon, il doit poursuivre avec le typage du fichier source. Lorsqu'une erreur de typage est détectée par le compilateur, elle doit être signalée le plus précisément possible, de la manière suivante :

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

2. à Jean-Christophe.Filliatre@cnrs.fr
3. Le correcteur est susceptible d'utiliser cet éditeur.

Là encore, la nature du message est laissée à votre discrétion, mais la forme de la localisation est imposée. Le compilateur doit alors terminer avec le code de sortie 1 (`exit 1`). Si en revanche il n'y a pas d'erreur de typage, le compilateur doit terminer avec le code de sortie 0. En cas d'erreur du compilateur lui-même, le compilateur doit terminer avec le code de sortie 2 (`exit 2`). L'option `--type-only` indique de stopper la compilation après l'analyse sémantique (typage). Elle est donc sans effet dans cette première partie, mais elle doit être honorée néanmoins.

Partie 2 (à rendre pour le dimanche 19 janvier 18:00). Si le fichier d'entrée est conforme à la syntaxe et au typage de ce sujet, votre compilateur doit produire du code x86-64 et terminer avec le code de sortie 0, sans rien afficher. Si le fichier d'entrée est `file.koka`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.koka`). Ce fichier x86-64 doit pouvoir être exécuté avec la commande

```
gcc -no-pie file.s -o file
./file
```

Le résultat affiché sur la sortie standard doit être identique à celui donné par la commande suivante :

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

Remarque importante. La correction du projet est réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec l'instruction `println`, qui sont compilés avec votre compilateur et dont la sortie est comparée à la sortie attendue. (Voir les tests fournis dans le sous-répertoire `exec/`.) Il est donc très important de correctement compiler les appels à `println`.

Conseils. Il est fortement conseillé de procéder construction par construction que ce soit pour le typage ou pour la production de code, dans cet ordre : affichage (`println`), arithmétique, variables locales, `if`, fonctions, fonctions anonymes.