École Normale Supérieure 2018–2019

projet de compilation

Petit Kotlin

version 2 — 22 octobre 2018

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Kotlin, appelé Petit Kotlin par la suite, produisant du code x86-64. Il s'agit d'un fragment relativement petit du langage Kotlin, avec parfois même quelques petites incompatibilités. Néanmoins, votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Petit Kotlin mais corrects au sens de Kotlin. Le présent sujet décrit la syntaxe et le typage de Petit Kotlin, 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)
(...)	parenthésage
... | ...	alternative

Attention à ne pas confondre « * » et « $^+$ » avec « `*` » et « `+` » qui sont des symboles du langage Kotlin. De même, attention à ne pas confondre les parenthèses avec les terminaux `(` et `)`.

1.1 Conventions lexicales

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires peuvent prendre deux formes :

— débutant par `/*` et s'étendant jusqu'à `*/`, et pouvant 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 :

$$\begin{array}{lcl} \langle chiffre \rangle & ::= & \texttt{0–9} \\ \langle alpha \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \mid \texttt{\_} \\ \langle ident \rangle & ::= & \langle alpha \rangle\ (\langle alpha \rangle \mid \langle chiffre \rangle)^\star \end{array}$$

Les identificateurs suivants sont des mots clés :

```
class    data    else    false   fun
if       null    return  this    true
val      var     while
```

Les constantes obéissent aux expressions régulières ⟨*entier*⟩ et ⟨*chaîne*⟩ suivantes :

⟨*bit*⟩ ::= `0` | `1`
⟨*hexa*⟩ ::= `0`–`9` | `a`–`f` | `A`–`F`
⟨*entier*⟩ ::= ⟨*chiffre*⟩ | ⟨*chiffre*⟩ (⟨*chiffre*⟩ | `_`)* ⟨*chiffre*⟩
| (`0x` | `0X`) (⟨*hexa*⟩ | ⟨*hexa*⟩ (⟨*hexa*⟩ | `_`)* ⟨*hexa*⟩)
| (`0b` | `0B`) (⟨*bit*⟩ | ⟨*bit*⟩ (⟨*bit*⟩ | `_`)* ⟨*bit*⟩)
⟨*car*⟩ ::= tout caractère de code ASCII compris entre 32 et 126 (inclus), autre que `\` et `"`
| `\\` | `\"` | `\n` | `\t`
⟨*chaîne*⟩ ::= `"` ⟨*car*⟩* `"`

Les constantes entières doivent être comprises entre -2^{31} et $2^{31} - 1$.

1.2 Syntaxe

La grammaire des fichiers sources considérée est donnée figures 1 et 2. Le point d'entrée est le non-terminal ⟨*fichier*⟩.

⟨*fichier*⟩ ::= ⟨*decl*⟩* `EOF`
⟨*decl*⟩ ::= ⟨*var*⟩ `;` | ⟨*classe*⟩ | ⟨*fonction*⟩
⟨*var*⟩ ::= (`var` | `val`) ⟨*ident*⟩ (`:` ⟨*type*⟩)? `=` ⟨*expr*⟩
⟨*classe*⟩ ::= `data class` ⟨*ident*⟩ (`<` ⟨*ident*⟩$^+_,$ `>`)?
`(` ⟨*paramètre_c*⟩$^+_,$ `)` ({ (⟨*var*⟩$^+_;$ `;`?)? })?
⟨*fonction*⟩ ::= `fun` (`<` ⟨*ident*⟩$^+_,$ `>`)? ⟨*ident*⟩
`(` ⟨*paramètre*⟩$^\star_,$ `)` (`:` ⟨*type*⟩)? ⟨*bloc*⟩
⟨*paramètre*⟩ ::= ⟨*ident*⟩ `:` ⟨*type*⟩
⟨*paramètre_c*⟩ ::= (`var` | `val`) ⟨*ident*⟩ `:` ⟨*type*⟩
⟨*type*⟩ ::= ⟨*ident*⟩ (`<` ⟨*type*⟩$^+_,$ `>`)?
| ⟨*type*⟩ `?`
| `(` ⟨*type*⟩ `)`
| `(` ⟨*type*⟩$^\star_,$ `)` `->` ⟨*type*⟩ V2

FIGURE 1 – Grammaire des fichiers de Petit Kotlin.

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle entier \rangle \mid \langle chaîne \rangle \mid \texttt{true} \mid \texttt{false} \\
& \mid & \texttt{this} \mid \texttt{null} \\
& \mid & \texttt{(}\ \langle expr \rangle\ \texttt{)} \\
& \mid & \langle accès \rangle \\
& \mid & \langle accès \rangle\ \texttt{=}\ \langle expr \rangle \\
& \mid & \langle ident \rangle\ \texttt{(}\ \langle expr \rangle^{\star}_{,}\ \texttt{)} \\
& \mid & \texttt{!}\ \langle expr \rangle \mid \texttt{-}\ \langle expr \rangle \\
& \mid & \langle expr \rangle\ \langle opérateur \rangle\ \langle expr \rangle \\
& \mid & \texttt{if (}\ \langle expr \rangle\ \texttt{)}\ \langle blocexpr \rangle\ (\texttt{else}\ \langle blocexpr \rangle)? \\
& \mid & \texttt{while (}\ \langle expr \rangle\ \texttt{)}\ \langle blocexpr \rangle \\
& \mid & \texttt{return}\ \langle expr \rangle? \\
& \mid & \texttt{fun (}\ \langle paramètre \rangle^{\star}_{,}\ \texttt{)}\ (\texttt{:}\ \langle type \rangle)?\ \langle bloc \rangle \\
\langle bloc \rangle & ::= & \{\ ((\langle var \rangle \mid \langle expr \rangle)^{+}_{;}\ ;?)?\ \} \\
\langle blocexpr \rangle & ::= & \langle bloc \rangle \mid \langle expr \rangle \\
\langle opérateur \rangle & ::= & \texttt{===} \mid \texttt{!==} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
& \mid & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \\
\langle accès \rangle & ::= & \langle ident \rangle \mid \langle expr \rangle\ \texttt{.}\ \langle ident \rangle \mid \langle expr \rangle\ \texttt{?.}\ \langle ident \rangle
\end{array}
$$

FIGURE 2 – Grammaire des expressions de Petit Kotlin.

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`	
`while`, `return`	
`=`	droite
`||`	gauche
`&&`	gauche
`===`, `!==`, `==`, `!=`	gauche
`>`, `>=`, `<`, `<=`	gauche
`+`, `-`	gauche
`*`, `/`, `%`	gauche
`-` (unaire), `!`	droite
`.`, `?.`	gauche
`->`	droite
`?`	

Par ailleurs, le `else` est toujours associé au `if` le plus proche.

Sucre syntaxique. On a les équivalences suivantes :
- le type τ`??` équivaut au type τ`?`.
- l'expression `if (`e_1`)` e_2 équivaut à `if (`e_1`)` e_2 `else {}`.
- une déclaration de fonction sans type de retour `fun` m`<...>(...){...}` équivaut à `fun` m`<...>(...):Unit {...}`. De même pour une fonction anonyme.

2 Typage statique

Une fois l'analyse syntaxique effectuée avec succès, on vérifie la conformité du fichier source. Dans tout ce qui suit, les types sont de la forme suivante :

$$\begin{array}{rcl} \tau & ::= & C\langle \tau, \ldots, \tau \rangle \\ & | & \tau\texttt{?} \\ & | & \texttt{Null} \\ & | & (\tau, \ldots, \tau) \to \tau \end{array}$$

où C désigne une classe. Un type $C\langle \tau_1, \ldots, \tau_n \rangle$ définit naturellement une substitution σ des paramètres de types $T_1, \ldots, T_n$ de la classe C par les types effectifs $\tau_1, \ldots, \tau_n$. Par la suite, on utilisera parfois la notation $C\langle\sigma\rangle$. Un contexte de typage Γ contient un ensemble de classes, de fonctions et de variables de la forme `var` $x : \tau$ (variable mutable) ou `val` $x : \tau$ (variable immuable). Les classes suivantes sont prédéfinies : `Boolean`, `Int`, `Unit`, `String` et `Array`$\langle\tau\rangle$.

Bonne formation d'un type. Le jugement $\Gamma \vdash \tau\ bf$ signifie « le type τ est bien formé dans l'environnement Γ ». Il est défini ainsi :

$$\frac{}{\Gamma \vdash \texttt{Null}\ bf} \qquad \frac{\Gamma \vdash \tau\ bf}{\Gamma \vdash \tau\texttt{?}\ bf} \qquad \frac{\forall i,\ \Gamma \vdash \tau_i\ bf \quad \Gamma \vdash \tau\ bf}{\Gamma \vdash (\tau_1, \ldots, \tau_n) \to \tau\ bf}$$

$$\frac{C\langle T_1, \ldots, T_n \rangle \in \Gamma \qquad \forall i, \Gamma \vdash \tau_i\ bf}{\Gamma \vdash C\langle \tau_1, \ldots, \tau_n \rangle\ bf}$$

Dit autrement, un type est bien formé s'il existe et si ses arguments effectifs sont eux-mêmes des types bien formés.

Champs et constructeur d'une classe. On note $C\{\ v\ x : \tau\ \}$ le fait que la classe C possède un champ x de type τ, où v désigne `var` (champ mutable) ou `val` (champ immuable). Ce champ peut être l'un des arguments du constructeur ou un champ additionnel introduit dans le corps de la classe C. On note $C\langle T_1, \ldots, T_k\rangle(\tau_1, \ldots, \tau_n)$ le fait que le constructeur de la classe C prend des arguments de types $\tau_1, \ldots, \tau_n$.

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

$$\frac{\Gamma \vdash e : \texttt{Null}}{\Gamma \vdash e : C\langle\sigma\rangle\texttt{?}} \qquad \frac{\Gamma \vdash e : C\langle\sigma\rangle}{\Gamma \vdash e : C\langle\sigma\rangle\texttt{?}}$$

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{}{\Gamma \vdash \texttt{null} : \texttt{Null}} \qquad \frac{\texttt{this} : \tau \in \Gamma}{\Gamma \vdash \texttt{this} : \tau} \qquad \frac{v\ x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : C\langle\sigma\rangle \quad C\{\ v\ x : \tau\ \}}{\Gamma \vdash e.x : \sigma(\tau)} \qquad \frac{\Gamma \vdash e : C\langle\sigma\rangle\texttt{?} \quad C\{\ v\ x : \tau\ \}}{\Gamma \vdash e\texttt{?}.x : \sigma(\tau)\texttt{?}}$$

$$\frac{\texttt{var}\ x : \tau \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash x\ \texttt{=}\ e : \texttt{Unit}}$$

$$\frac{\Gamma \vdash e : C\langle\sigma\rangle \quad C\{\ \texttt{var}\ x : \tau\ \} \quad \Gamma \vdash e_2 : \sigma(\tau)}{\Gamma \vdash e.x\ \texttt{=}\ e_2 : \texttt{Unit}}$$

$$\frac{\Gamma \vdash e : C\langle\sigma\rangle\texttt{?} \quad C\{\ \texttt{var}\ x : \tau\ \} \quad \Gamma \vdash e_2 : \sigma(\tau)}{\Gamma \vdash e\texttt{?}.x\ \texttt{=}\ e_2 : \texttt{Unit}}$$

$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{-}\ e : \texttt{Int}} \qquad \frac{\Gamma \vdash e : \texttt{Boolean}}{\Gamma \vdash \texttt{!}e : \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 \notin \{\texttt{Unit}, \texttt{Boolean}, \texttt{Int}\} \quad op \in \{\texttt{===},\ \texttt{!==}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int} \quad op \in \{\texttt{==},\ \texttt{!=},\ \texttt{<},\ \texttt{<=},\ \texttt{>},\ \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int} \quad op \in \{\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{\%}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Boolean} \quad \Gamma \vdash e_2 : \texttt{Boolean} \quad op \in \{\texttt{\&\&},\ \texttt{||}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e : \texttt{String?}}{\Gamma \vdash \texttt{print}(e) : \texttt{Unit}} \qquad \frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{print}(e) : \texttt{Unit}}$$

$$\frac{\Gamma \vdash e : \texttt{Boolean} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}(e)\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Boolean} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{while}(e_1)\ e_2 : \texttt{Unit}}$$

$$\frac{C\langle\ldots\rangle(\tau_1, \ldots, \tau_n) \in \Gamma \quad \forall i,\ \Gamma \vdash e_i : \sigma(\tau_i)}{\Gamma \vdash C(e_1, \ldots, e_n) : C\langle\sigma\rangle}$$

$$\frac{f\langle\ldots\rangle(\tau_1, \ldots, \tau_n) : \tau \in \Gamma \quad \forall i,\ \Gamma \vdash e_i : \sigma(\tau_i)}{\Gamma \vdash f(e_1, \ldots, e_n) : \sigma(\tau)}$$

Dans une application de constructeur ou de fonction, tous les paramètres de type doivent être résolus au moment de l'appel. V2

$$\frac{v\ f : (\tau_1, \ldots, \tau_n) \to \tau \in \Gamma \quad \forall i,\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

$$\frac{\tau_r = \texttt{Unit}}{\Gamma \vdash \texttt{return} : \texttt{Unit}} \qquad \frac{\Gamma \vdash e : \tau_r}{\Gamma \vdash \texttt{return}\ e : \texttt{Unit}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma + v\ x : \tau \vdash \{e_2; \ldots; e_n\} : \tau_2}{\Gamma \vdash \{v\ x = e; e_2; \ldots; e_n\} : \tau_2}$$

$$\frac{\Gamma \vdash \tau\ bf \quad \Gamma \vdash e : \tau \quad \Gamma + v\ x : \tau \vdash \{e_2; \ldots; e_n\} : \tau_2}{\Gamma \vdash \{v\ x : \tau = e; e_2; \ldots; e_n\} : \tau_2}$$

$$\frac{}{\Gamma \vdash \{\} : \texttt{Unit}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \tau} \qquad \frac{n \geq 2 \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \{e_2; \ldots; e_n\} : \tau_2}{\Gamma \vdash \{e_1; \ldots; e_n\} : \tau_2}$$

Par ailleurs, toutes les variables introduites dans un *même* bloc doivent porter des noms différents.

Test de non-nullité. Quand le contexte permet d'être certain qu'une *variable* ne peut avoir la valeur `null`, alors on s'autorise à lui donner un type plus précis avec la règle suivante :

$$\frac{\Gamma \vdash x : C\langle\sigma\rangle\texttt{?} \quad x \text{ ne peut être } \texttt{null}}{\Gamma \vdash x : C\langle\sigma\rangle}$$

Un exemple de tel contexte est celui d'un test explicite `if (`x `!== null)` ... mais il y en a d'autres. Les tests fournis illustrent plusieurs cas de figure, sans être exhaustifs.

Typage d'une classe. Soit Γ un environnement de typage. Pour y ajouter la déclaration d'une nouvelle classe C de la forme

$$\texttt{data class}\ C\langle T_1, \ldots, T_k\rangle(v_1\ x_1 : \tau_1, \ldots, v_n\ x_n : \tau_n)\ \{d_1; \ldots; d_m\}$$

on procède de la manière suivante :

1. On vérifie que les T_i sont deux à deux distincts. De même, on vérifie que les noms des champs (x_i et d_i) sont deux à deux distincts.
2. On construit un nouvel environnement Γ' en ajoutant à Γ tous les T_i comme autant de nouvelles classes, ainsi que la classe C.
3. On vérifie $\forall i$, $\Gamma' \vdash \tau_i\ bf$ et on ajoute à Γ' tous les $v_i\ x_i : \tau_i$, ainsi que `val this` : $C\langle T_1, \ldots, T_k\rangle$.
4. On vérifie enfin les déclarations $d_1, \ldots, d_m$ dans Γ'. Pour une déclaration de $v\ x = e$ ou $v\ x : \tau = e$ on procède comme pour une variable locale, puis on ajoute le champ x à la classe C.

Typage d'une fonction. Soit Γ un environnement de typage. Pour y ajouter la déclaration de fonction

$$\texttt{fun}\langle T_1, \ldots, T_n\rangle\ f(x_1 : \tau_1, \ldots, x_m : \tau_m) : \tau\{b\}$$

on procède de la manière suivante :

1. On vérifie que les T_i sont deux à deux distincts. De même, on vérifie que les noms des paramètres x_i sont deux à deux distincts.
2. On construit un nouvel environnement Γ' en ajoutant à Γ tous les T_i comme autant de nouvelles classes.
3. On vérifie $\forall i$, $\Gamma' \vdash \tau_i\ bf$ ainsi que $\Gamma' \vdash \tau\ bf$, et on ajoute à Γ' tous les `val` $x_i : \tau_i$.
4. On ajoute la fonction f à Γ' (pour permettre la récursivité).
5. Enfin, on type le bloc b dans Γ' avec $\tau_r = \tau$ (type de retour). En outre, on vérifie la présence d'une instruction `return` dans chaque branche possible de l'exécution dès lors que $\tau \neq$ `Unit`.

On procède d'une manière similaire pour une fonction anonyme, à laquelle on donne un type de la façon suivante :

$$\frac{\forall i, \Gamma \vdash \tau_i\ bf \quad \Gamma \vdash \tau\ bf \quad \Gamma + x_1 : \tau_1 + \cdots + x_n : \tau_n \vdash \{b\} : \tau'}{\Gamma \vdash \texttt{fun}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\{b\} : (\tau_1, \ldots, \tau_n) \to \tau}$$

On note qu'une fonction anonyme ne peut être polymorphe.

Typage d'un fichier. Pour typer un fichier, on construit progressivement un environnement en typant les déclarations contenues dans le fichier, dans l'ordre où elles apparaissent. (Cela veut dire notamment qu'il n'y a pas de récursion mutuelle en Petit Kotlin, à la différence de Kotlin.) Pour une déclaration de variable globale, on procède comme pour une variable locale. Pour une classe ou une fonction, on procède comme expliqué ci-dessus. On vérifie par ailleurs l'unicité des noms de classes, des noms de fonctions et des noms de variables globales sur l'ensemble du fichier.

Enfin, on vérifie qu'il existe une fonction `main` sans paramètres de types, avec un unique paramètre de type `Array<String>` et avec le type de retour `Unit`. La classe `Array` est une classe dont on ne sait rien.

Anticipation. Dans la phase suivante (production de code), certaines informations provenant du typage seront nécessaires, telles que par exemple la portée, la détermination de la fonction appelée, etc. 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 plus d'information lorsque c'est nécessaire. (Cf le cours 4.)

3 Production de code

L'objectif est de réaliser un compilateur simple mais correct. En particulier, on ne cherche pas à faire d'allocation de registres mais on se contente d'utiliser la pile pour stocker les éventuels calculs intermédiaires. Bien entendu, il est possible d'utiliser localement les registres. On ne cherche pas à libérer la mémoire.

3.1 Représentation des valeurs

On choisit une représentation simple des valeurs, où chaque valeur occupe exactement un mot de 8 octets (64 bits).

Types primitifs. On suppose qu'on ne manipule jamais de valeur de type `Int?`. Du coup, on peut représenter un entier directement par un entier machine, en l'occurrence un entier 64 bits signé. Mais attention : un entier doit être affiché (avec `print`) comme un entier 32 bits signé. Les booléens sont représentés par des entiers : 0 représente `false` et `1` représente `true`. La valeur du type `Unit` est représentée par l'entier 0.

Objets. Un objet est représenté par l'adresse d'un bloc alloué sur le tas. Ce bloc contient les valeurs des champs de l'objet, dans un ordre arbitraire. Le compilateur maintient donc une table donnant, pour chaque classe C et chaque champ de C, la position où trouver ce champ dans un objet de la classe C. Il est inutile de stocker la classe C dans l'objet car il n'y a pas d'appel de méthode dans Petit Kotlin.

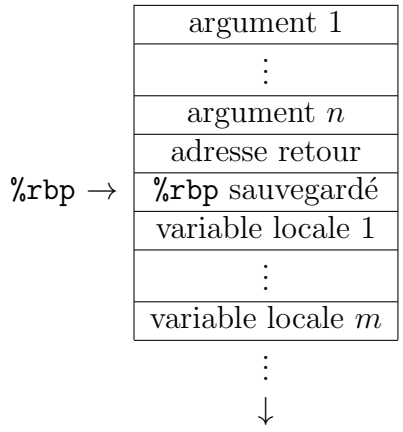
La valeur `null`. Elle est représentée par l'entier 0. En particulier, elle est différente de toute adresse d'un objet alloué sur le tas.

Chaînes de caractères. Un objet de la classe `String` est représenté par l'adresse d'une chaîne de caractères allouée dans le segment de données.

Fonctions de première classe (lambdas). Une fonction de première classe (*i.e.* une valeur obtenue avec une expression `fun`) est représentée par l'adresse d'un bloc alloué sur le tas contenant $n + 1$ mots décrivant une fermeture, c'est-à-dire un premier mot contenant l'adresse du code à appeler et les suivants contenant les valeurs des variables capturées dans cette fermeture. (Cf le cours 9.)

3.2 Schéma de compilation

L'appelant place tous les arguments sur la pile avant de faire `call`. L'appelé sauvegarde `%rbp` sur la pile et le positionne à cet endroit. Il alloue éventuellement de la place sur la pile pour ses variables locales. La valeur de retour est placée dans le registre `%rax`. Au retour, l'appelant se charge de dépiler les arguments. On a donc un tableau d'activation de la forme suivante :



Pour une fonction de première classe, la fermeture est passée comme un argument supplémentaire (par exemple le premier). Pour un constructeur, l'objet (déjà alloué) est passé comme un argument supplémentaire (par exemple le premier). La valeur d'une expression est compilée en utilisant la pile si besoin et en plaçant sa valeur finale dans `%rdi` ou en sommet de pile (au choix).

Le code assembleur produit au final doit ressembler à quelque chose comme

```
        .text
        .globl main
main:   initialisation des variables globales
        appel de la fonction main
        xorq %rax, %rax
        ret
...     fonctions (globales, de première classe
...     et constructeurs)
        .data
...     variables globales
...     chaînes de caractères
```

4 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`, sous la forme d'une archive `tar` compressée (option "`z`" de `tar`), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). 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 pkotlinc. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés 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 `ocamlbuild` ou `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 ASCII, Markdown ou PDF.

Partie 1 (à rendre pour le dimanche 9 décembre 18:00). Dans cette première partie du projet, le compilateur pkotlinc doit accepter sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Petit Kotlin portant l'extension `.kt`. 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.kt", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs 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.kt", line 4, characters 5-6:
this expression has type Int but is expected to have type Unit
```

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.

Partie 2 (à rendre pour le mercredi 9 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.kt`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.kt`). Ce fichier x86-64 doit pouvoir être exécuté avec la commande

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

Le résultat affiché sur la sortie standard doit être identique à celui donné par l'exécution du fichier Kotlin `file.kt`, par exemple avec

```
kotlinc file.kt -include-runtime -d exec.jar
java -jar exec.jar
```

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 `print`, qui sont compilés avec votre compilateur et dont la sortie est comparée à la sortie attendue. Il est donc très important de correctement compiler les appels à `print`.

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, arithmétique, variables locales, fonctions (globales et monomorphes), classes (monomorphes), polymorphisme, fonctions de première classe.