École Normale Supérieure 2015–2016

projet de compilation

Petit Scala

partie 1 — à rendre avant le dimanche 6 décembre 18:00

version 2 — 3 décembre 2015

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Scala, appelé Petit Scala par la suite, produisant du code X86-64. Il s'agit d'un fragment relativement petit du langage Scala, avec parfois même quelques petites incompatibilités. Néanmoins, votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Petit Scala mais corrects au sens de Scala. Le présent sujet décrit la syntaxe et le typage de Petit Scala, ainsi que la nature du travail demandé dans cette première partie.

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 Scala. 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 ne 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 :

$\langle chiffre \rangle$ ::= `0`–`9`
$\langle alpha \rangle$ ::= `a`–`z` | `A`–`Z`
$\langle ident \rangle$ ::= $\langle alpha \rangle$ ($\langle alpha \rangle$ | $\langle chiffre \rangle$ | `_`)*

Les identificateurs suivants sont des mots clés :

```
class    def     else    eq      extends   false
if       ne      new     null    object    override
print    return  this    true    val       var
while
```

Les constantes obéissent aux expressions régulières $\langle entier \rangle$ et $\langle chaîne \rangle$ suivantes :

$\langle entier \rangle$ ::= `0` | `1`–`9` $\langle chiffre \rangle^\star$
$\langle car \rangle$ ::= tout caractère de code ASCII compris entre 32 et 126 (inclus), autre que `\` et `"`
| `\\` | `\"` | `\n` | `\t`
$\langle chaîne \rangle$ ::= `"` $\langle car \rangle^\star$ `"`

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 $\langle fichier \rangle$.

$\langle fichier \rangle$ ::= $\langle classe \rangle^\star$ $\langle classe\_Main \rangle$ `EOF`
$\langle classe \rangle$ ::= `class` $\langle ident \rangle$ (`[` $\langle param\_type\_classe \rangle^+_,$ `]`)?
(`(` $\langle paramètre \rangle^\star_,$ `)`)?
(`extends` $\langle type \rangle$ (`(` $\langle expr \rangle^\star_,$ `)`)?)? `{` $\langle decl \rangle^\star_;$ `}`
$\langle decl \rangle$ ::= $\langle var \rangle$ | $\langle méthode \rangle$

$\langle var \rangle$ ::= (`var` | `val`) $\langle ident \rangle$ (`:` $\langle type \rangle$)? `=` $\langle expr \rangle$
$\langle méthode \rangle$ ::= `override`? `def` $\langle ident \rangle$ (`[` $\langle param\_type \rangle^+_,$ `]`)?
`(` $\langle paramètre \rangle^\star_,$ `)` ($\langle bloc \rangle$ | `:` $\langle type \rangle$ `=` $\langle expr \rangle$)

$\langle paramètre \rangle$ ::= $\langle ident \rangle$ `:` $\langle type \rangle$
$\langle param\_type \rangle$ ::= $\langle ident \rangle$ ((`<:` | `>:`) $\langle type \rangle$)?
$\langle param\_type\_classe \rangle$ ::= (`+` | `-`)? $\langle param\_type \rangle$
$\langle type \rangle$ ::= $\langle ident \rangle$ $\langle arguments\_type \rangle$
$\langle arguments\_type \rangle$ ::= (`[` $\langle type \rangle^+_,$ `]`)?

$\langle classe\_Main \rangle$ ::= `object Main` `{` $\langle decl \rangle^\star_;$ `}`

FIGURE 1 – Grammaire des fichiers de Petit Scala.

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

FIGURE 2 – Grammaire des expressions de Petit Scala.

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
`eq`, `ne`, `==`, `!=`	gauche
`>`, `>=`, `<`, `<=`	gauche
`+`, `-`	gauche
`*`, `/`, `%`	gauche
`-` (unaire), `!`	droite
`.`	gauche

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

Sucre syntaxique. On a les équivalences suivantes :
- l'expression `if (`e_1`)` e_2 équivaut à `if (`e_1`)` e_2 `else ()`.
- un appel de méthode m`[`$\tau_1, \ldots \tau_k$`](`$e_1, \ldots, e_n$`)` sans objet explicite équivaut à un appel `this.`m`[`$\tau_1, \ldots, \tau_k$`](`$e_1, \ldots, e_n$`)`.
- une déclaration de méthode sans type de retour `def` m`[...](...)`$\langle bloc \rangle$ équivaut à `def` m`[...](...):Unit =` $\langle bloc \rangle$.

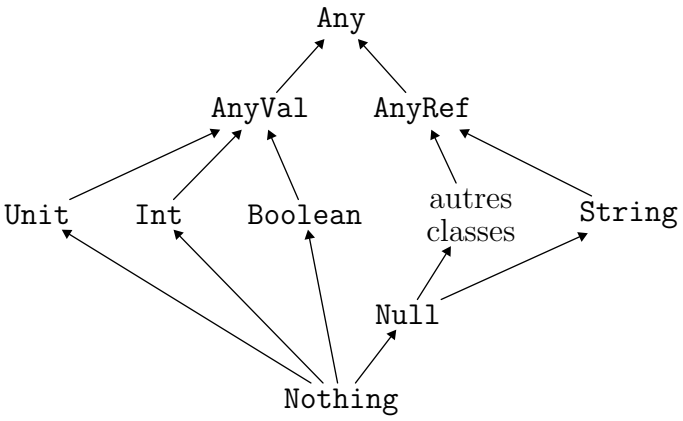

FIGURE 3 – Diagramme de classes.

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 :

$$\tau ::= C[\tau, \ldots, \tau]$$

où C désigne une classe. Un type $C[\tau_1, \ldots, \tau_n]$ 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 $C[\sigma]$. Un contexte de typage Γ contient

- un ensemble de classes ;
- un ensemble de contraintes de types C`>:`τ ;
- une suite ordonnée de déclarations de variables de la forme `var` $x : \tau$ (variable mutable) ou `val` $x : \tau$ (variable immuable).

Les classes suivantes sont prédéfinies : `Any`, `AnyVal`, `Boolean`, `Int`, `Unit`, `AnyRef`, `String`, `Null` et `Nothing`. Toute nouvelle classe est une super-classe de `Null`. D'autre part, si elle n'est pas déclarée explicitement comme héritant d'une autre classe, alors elle hérite de la classe `AnyRef`. On note $C_1 \longrightarrow C_2$ la relation « la classe C_1 est une sous-classe de la classe C_2 ». Le diagramme de classes est donné figure 3.

2.1 Sous-typage

La relation de sous-typage $\Gamma \vdash \tau_1 \leq \tau_2$ signifie « le type τ_1 est un sous-type du type τ_2 ». Intuitivement, on peut l'interpréter par « toute valeur de type τ_1 peut être donnée là où est attendue une valeur de type τ_2 ». Elle est définie ainsi :

$$\frac{}{\Gamma \vdash \texttt{Nothing} \leq \tau} \qquad \frac{\texttt{Null} \longrightarrow C_2}{\Gamma \vdash \texttt{Null} \leq C_2[\sigma_2]} \qquad \frac{\begin{array}{rl} & C[\ldots, \texttt{+}T_i, \ldots] \Rightarrow \Gamma \vdash \tau_i \leq \tau_i' \\ \forall i, & C[\ldots, T_i, \ldots] \ \ \Rightarrow \tau_i = \tau_i' \\ & C[\ldots, \texttt{-}T_i, \ldots] \Rightarrow \Gamma \vdash \tau_i' \leq \tau_i \end{array}}{\Gamma \vdash C[\tau_1, \ldots, \tau_n] \leq C[\tau_1', \ldots, \tau_n']}$$

$$\frac{C_1 \texttt{ extends } C[\tau_1, \ldots, \tau_n] \quad C \longrightarrow C_2 \quad C[\sigma_1(\tau_1), \ldots, \sigma_1(\tau_n)] \leq C_2[\sigma_2]}{\Gamma \vdash C_1[\sigma_1] \leq C_2[\sigma_2]}$$

$$\frac{C_1 \not\rightarrow C_2 \quad C_2\texttt{>:}\tau \in \Gamma \quad \Gamma \vdash C_1[\sigma_1] \leq \tau}{\Gamma \vdash C_1[\sigma_1] \leq C_2}$$

2.2 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 :

$$\frac{C[T_1, \ldots, T_n] \in \Gamma \quad \forall i, \Gamma \vdash \tau_i\ bf \quad \Gamma \vdash \{T_i \mapsto \tau_i\}\ bf}{\Gamma \vdash C[\tau_1, \ldots, \tau_n]\ bf}$$

où les T_i sont les paramètres de types et $\{T_i \mapsto \tau_i\}$ la substitution de ces paramètres par les types effectifs τ_i. Une telle substitution σ est bien formée si elle respecte les bornes associées à chaque paramètre, le cas échéant. Si on note $co(T)$ la borne du paramètre T, alors la bonne formation de σ est définie ainsi :

$$\frac{\forall i, \begin{array}{l} co(T_i) = \texttt{<:}\tau \Rightarrow \Gamma \vdash \sigma(T_i) \leq \sigma(\tau) \\ co(T_i) = \texttt{>:}\tau \Rightarrow \Gamma \vdash \sigma(\tau) \leq \sigma(T_i) \end{array}}{\Gamma \vdash \sigma\ bf}$$

Dit autrement, un type est bien formé si les arguments effectifs sont eux-mêmes des types bien formés et si les bornes sur les paramètres sont respectées.

2.3 Champs, constructeurs et méthodes 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 déclaré dans C ou hérité d'une super-classe de C. On note $C(\tau_1, \ldots, \tau_n)$ le fait que le constructeur de la classe C prend des arguments de types $\tau_1, \ldots, \tau_n$. On note $C\{\ m[T_1, \ldots, T_k](\tau_1, \ldots, \tau_n) : \tau\ \}$ le fait que la classe C possède une méthode $m[T_1, \ldots, T_k]$ prenant des arguments de type $\tau_1, \ldots, \tau_n$ et renvoyant une valeur de type τ. Cette méthode peut être déclarée dans C ou héritée d'une super-classe de C.

Dans la suite, le type τ_r désigne le type de retour de la méthode que l'on est en train de typer, le cas échéant.

2.4 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 :

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

$$\frac{x \notin \Gamma \quad \Gamma \vdash \texttt{this}.x : \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : C[\sigma] \quad C\{\ v\ x : \tau\ \}}{\Gamma \vdash e.x : \sigma(\tau)}$$

$$\frac{\texttt{var}\ x : \tau_1 \in \Gamma \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_1}{\Gamma \vdash x\ \texttt{=}\ e_2 : \texttt{Unit}}$$

$$\frac{x \notin \Gamma \quad \Gamma \vdash \texttt{this}.x = e_2 : \texttt{Unit}}{\Gamma \vdash x \texttt{ = } e_2 : \texttt{Unit}}$$

$$\frac{\Gamma \vdash e : C[\sigma] \quad C\{\ \texttt{var}\ x : \tau_1\ \} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash \tau_2 \leq \sigma(\tau_1)}{\Gamma \vdash e.x \texttt{ = } e_2 : \texttt{Unit}}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \leq \texttt{AnyRef} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \texttt{AnyRef} \quad op \in \{\texttt{eq}, \texttt{ne}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Boolean}}$$

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

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

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

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

$$\frac{\Gamma \vdash e : \texttt{Boolean} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \leq \tau_2 \vee \tau_2 \leq \tau_1}{\Gamma \vdash \texttt{if}(e)\ e_1\ \texttt{else}\ e_2 : \max(\tau_1, \tau_2)}$$

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

$$\frac{\Gamma \vdash C[\sigma]\ bf \quad C(\tau_1, \ldots, \tau_n) \quad \forall i,\ \Gamma \vdash e_i : \tau_i' \quad \Gamma \vdash \tau_i' \leq \sigma(\tau_i)}{\Gamma \vdash \texttt{new}\ C[\sigma](e_1, \ldots, e_n) : C[\sigma]}$$

V2

$$\frac{\begin{array}{c}\Gamma \vdash e : C[\sigma] \quad C\{\ m[T_1, \ldots, T_k](\tau_1', \ldots, \tau_n') : \tau\ \} \quad \forall j,\ \Gamma \vdash \tau_j\ bf \\ \sigma' := \{T_j \mapsto \tau_j\} \quad \Gamma \vdash \sigma \circ \sigma'\ bf \quad \forall i,\ \Gamma \vdash e_i : \tau_i'' \quad \Gamma \vdash \tau_i'' \leq \sigma \circ \sigma'(\tau_i')\end{array}}{\Gamma \vdash e.m[\tau_1, \ldots, \tau_k](e_1, \ldots, e_n) : \sigma \circ \sigma'(\tau)}$$

$$\frac{\Gamma \vdash \texttt{Unit} \leq \tau_r}{\Gamma \vdash \texttt{return} : \texttt{Nothing}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \leq \tau_r}{\Gamma \vdash \texttt{return}\ e : \texttt{Nothing}}$$

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

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

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

2.5 Typage d'une classe

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

$$\texttt{class}\ C[T_1, \ldots, T_k](x_1 : \tau_1, \ldots, x_n : \tau_n)\ \texttt{extends}\ \tau(e_1, \ldots, e_p)\ \{d_1; \ldots; d_m\}$$

on procède de la manière suivante :

1. On construit un nouvel environnement Γ' en étendant Γ de la manière suivante : pour chaque paramètre de type T_i, dans l'ordre,
   (a) on vérifie que sa borne, le cas échéant, est un type bien formé ;
   (b) on ajoute T_i comme une nouvelle classe ;
   (c) si on a une borne $T_i\texttt{<:}\tau_i$, on déclare que T_i hérite de τ_i ;
   (d) si on a une borne $T_i\texttt{>:}\tau_i$, on ajoute la contrainte $T_i\texttt{>:}\tau_i$.
2. On vérifie $\Gamma' \vdash \tau\ bf$ et on ajoute la classe C à Γ (avec uniquement les champs et méthodes hérités pour l'instant).
3. On vérifie $\forall i$, $\Gamma' \vdash \tau_i\ bf$ et on ajoute à Γ' tous les $\texttt{val}\ x_i : \tau_i$, ainsi que $\texttt{val this} : C[T_1, \ldots, T_k]$.
4. On vérifie $\Gamma' \vdash \texttt{new}\ \tau(e_1, \ldots, e_p) : \tau$, autrement dit, que l'appel au constructeur de la super classe est légal.
5. On vérifie les déclarations $d_1, \ldots, d_m$ dans Γ' :
   – pour une déclaration de champ $v\ x = e$ ou $v\ x : \tau = e$ on procède comme pour une variable locale, puis on ajoute le champ x à la classe C.
   – pour une déclaration de méthode

   $$m\texttt{[}T'_1, \ldots, T'_j\texttt{]}\,\texttt{(}x'_1 : \tau'_1, \ldots, x'_l : \tau'_l\texttt{)} : \tau = e$$

   on construit un nouvel environnement Γ'' en étendant Γ' avec les paramètres $T'_1, \ldots, T'_j$ (comme plus haut pour les paramètres de classes) ; on vérifie que les types $\tau'_1, \ldots, \tau'_l, \tau$ sont bien formés ; on ajoute la méthode m à la classe C ; on ajoute les variables x'_i à Γ'' ; enfin, on type l'expression e et on vérifie que son type est plus petit que τ.
   Par ailleurs, la définition doit être précédée du mot-clé `override` si et seulement si m est déjà définie dans une super-classe de C. Dans ce cas, les types de ses arguments doivent être les mêmes que dans la super-classe et le type de retour τ doit être plus petit que le type de retour de m dans la super-classe. Cette comparaison est faite au nom près des paramètres de types (α-équivalence). Les bornes sur les paramètres de types, lorsqu'elles existent, doivent être les mêmes.

Par ailleurs, on vérifiera que la classe singleton `Main` contient bien une méthode `main` sans paramètre de types, avec un paramètre de type `Array[String]` et avec le type de retour `Unit`. La classe `Array` est une classe paramétrée dont on ne sait rien.

Note : À la différence de Scala, il n'y a pas de récursion mutuelle entre classes, ni entre champs et méthodes à l'intérieur d'une classe.

2.6 Vérification de la variance

Les positions où un type peut apparaître se répartissent en trois catégories : positives, négatives et neutres. Un paramètre de type déclaré comme covariant (`+T` dans la syntaxe concrète) doit apparaître uniquement dans des positions positives et un paramètre de type déclaré comme contravariant (`-T` dans la syntaxe concrète) doit apparaître uniquement dans des positions négatives. Les positions positives et négatives sont ainsi définies :

- le type d'un champ mutable est une position neutre, celui d'un champ immuable est une position positive ;
- le type d'un argument de méthode est une position négative et le type de retour d'une méthode une position positive ;
- le type τ derrière `extends` est une position positive ;
- dans une contrainte `<:`τ (resp. `>:`τ) portant sur un paramètre de type de classe, le type τ est une position positive (resp. négative) ;
- dans une contrainte `<:`τ (resp. `>:`τ) portant sur un paramètre de type de méthode, le type τ est une position négative (resp. positive) ;
- si le type $C[\tau_1, \ldots, \tau_n]$ est en position positive (resp. négative) et que le i-ième paramètre de C est covariant, alors τ_i est une position positive (resp. négative) ;
- si le type $C[\tau_1, \ldots, \tau_n]$ est en position positive (resp. négative) et que le i-ième paramètre de C est contravariant, alors τ_i est une position négative (resp. positive).

Note : On n'effectue pas de vérification de variance au niveau des arguments des constructeurs. V2

2.7 Conditions d'existence et d'unicité

Enfin, les conditions suivantes doivent être vérifiées :

- chaque classe ne peut être définie qu'une seule fois ;
- une classe ne peut hériter d'une classe qui n'est pas encore définie, ni hériter d'une des classes `Any`, `AnyVal`, `Unit`, `Int`, `Boolean`, `String`, `Null`, ou `Nothing` ;
- tous les paramètres de types d'une même classe ou d'une même méthode doivent porter des noms différents ;
- tous les paramètres d'un constructeur ou d'une méthode doivent porter des noms différents ;
- tous les champs d'une même classe doivent porter des noms différents, qu'il s'agisse de champs hérités ou non, et ces noms doivent être différents des noms des paramètres du constructeur ;
- toutes les méthodes définies dans une *même* classe doivent porter des noms différents ;
- toutes les variables introduites dans un *même* bloc doivent porter des noms différents.

2.8 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 méthode 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.

3 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`. Votre projet doit se présenter sous 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* de votre compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé pscala. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources.

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, PostScript ou PDF.

Dans cette première partie du projet, votre compilateur pscala doit accepter sur sa ligne de commande une option éventuelle `--parse-only` et exactement un fichier Petit Scala portant l'extension `.scala`. 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.scala", 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, votre 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 votre compilateur, elle doit être signalée le plus précisément possible, de la manière suivante :

```
File "test.scala", 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`).

Remerciements. Merci à Martin Clochard pour son aide dans la préparation de ce sujet.