

projet de compilation

Petit Java

version 1 — 19 octobre 2011

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Java, appelé **Petit Java** par la suite, produisant du code MIPS. Il s'agit d'un fragment contenant des entiers, des booléens, des chaînes de caractères et des objets. Il s'agit d'un fragment 100% compatible avec Java, au sens où tout programme de **Petit Java** est aussi un programme Java correct. Ceci permettra notamment d'utiliser ce dernier comme référence. Le présent sujet décrit précisément **Petit Java**, ainsi que la nature du travail demandé.

1 Syntaxe

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

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

1.1 Conventions lexicales

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

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

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

$$\begin{aligned} \langle \text{chiffre} \rangle &::= 0-9 \\ \langle \text{alpha} \rangle &::= \text{a-z} \mid \text{A-Z} \\ \langle \text{ident} \rangle &::= (\langle \text{alpha} \rangle \mid _)(\langle \text{alpha} \rangle \mid \langle \text{chiffre} \rangle \mid _)^* \end{aligned}$$

Les identificateurs suivants sont des mots clés :

```
boolean    class    else    extends    false    for    if
instanceof int     new     null     public   return  static
this       true    void
```

Les constantes entières obéissent à l'expression régulière $\langle entier \rangle$ suivante :

$$\langle entier \rangle ::= 0 \mid 1-9 \langle chiffre \rangle^*$$

Les chaînes de caractères s'écrivent entre guillemets ("). Il y a deux séquences d'échappement : \backslash pour le caractère " et $\backslash n$ pour un retour-charriot. On suppose que les chaînes ne contiennent pas de caractères \backslash en dehors de ces séquences d'échappement.

1.2 Syntaxe

La grammaire des fichiers sources considérée est donnée figures 1 et 2. Le point d'entrée est le non-terminal $\langle fichier \rangle$.

```

 $\langle fichier \rangle$       ::=  $\langle classe \rangle^* \langle classe\_Main \rangle$  EOF
 $\langle classe \rangle$       ::= class  $\langle ident \rangle$  (extends  $\langle ident \rangle$ )? { decl* }
 $\langle decl \rangle$         ::=  $\langle type \rangle \langle ident \rangle$ ; |  $\langle constructeur \rangle$  |  $\langle méthode \rangle$ 

 $\langle constructeur \rangle$  ::=  $\langle ident \rangle$  (  $\langle paramètres \rangle$ ? ) {  $\langle instruction \rangle^*$  }
 $\langle méthode \rangle$      ::= (  $\langle type \rangle$  | void )  $\langle ident \rangle$  (  $\langle paramètres \rangle$ ? ) {  $\langle instruction \rangle^*$  }

 $\langle paramètres \rangle$  ::=  $\langle type \rangle \langle ident \rangle$  |  $\langle type \rangle \langle ident \rangle$  ,  $\langle paramètres \rangle$ 
 $\langle type \rangle$         ::= boolean | int |  $\langle ident \rangle$ 

 $\langle classe\_Main \rangle$  ::= class Main {
                        public static void main(String  $\langle ident \rangle$  []) {  $\langle instruction \rangle^*$  }
                        }

```

FIGURE 1 – Grammaire des fichiers de Petit Java.

Les associativités et précédences des diverses constructions sont données par la table suivante, de la plus faible à la plus forte précedence :

opérateur ou construction	associativité
=	droite
	gauche
&&	gauche
==, !=	gauche
>, >=, <, <=, instanceof	gauche
+, -	gauche
*, /, %	gauche
- (unaire), ++, --, !, cast	droite
.	gauche

```

⟨expr⟩ ::= ⟨entier⟩ | ⟨chaîne⟩ | true | false
        | this
        | null
        | ( ⟨expr⟩ )
        | ⟨accès⟩
        | ⟨accès⟩ = ⟨expr⟩
        | ⟨accès⟩ ( ⟨Lexpr⟩? )
        | new ⟨ident⟩ ( ⟨Lexpr⟩? )
        | ++ ⟨expr⟩ | -- ⟨expr⟩ | ⟨expr⟩ ++ | ⟨expr⟩ --
        | ! ⟨expr⟩ | - ⟨expr⟩
        | ⟨expr⟩ ⟨opérateur⟩ ⟨expr⟩
        | ( ⟨type⟩ ) ⟨expr⟩
        | ⟨expr⟩ instanceof ⟨type⟩

⟨opérateur⟩ ::= == | != | < | <= | > | >= | + | - | * | / | % | && | ||
⟨accès⟩ ::= ⟨ident⟩ | ⟨expr⟩ . ⟨ident⟩
⟨Lexpr⟩ ::= ⟨expr⟩ | ⟨expr⟩ , ⟨Lexpr⟩

⟨instruction⟩ ::= ;
               | ⟨expr⟩ ;
               | ⟨type⟩ ⟨ident⟩ ;
               | ⟨type⟩ ⟨ident⟩ = ⟨expr⟩ ;
               | if ( ⟨expr⟩ ) ⟨instruction⟩
               | if ( ⟨expr⟩ ) ⟨instruction⟩ else ⟨instruction⟩
               | for ( ⟨expr⟩? ; ⟨expr⟩? ; ⟨expr⟩? ) ⟨instruction⟩
               | { ⟨instruction⟩* }
               | return ⟨expr⟩? ;

```

FIGURE 2 – Grammaire des expressions et intructions de Petit Java.

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 :

$$\tau ::= \text{void} \mid \text{boolean} \mid \text{int} \mid C \mid \text{typenull}$$

où C désigne une classe. Il est pratique de considérer `void` comme un type, même si ce n'est pas un type au sens syntaxique. D'autre part, `typenull` est introduit pour pouvoir donner un type à l'expression `null`.

2.1 Héritage et sous-typage

On notera $C_1 \longrightarrow C_2$ la relation « la classe C_1 est une sous-classe de la classe C_2 », qui est la clôture réflexive-transitive de l'ensemble des déclarations d'héritage C_1 `extends` C_2 contenue dans le source.

Il y a deux classes prédéfinies, qui sont *Object* et *String*. Toute classe qui n'est pas déclarée comme héritant d'une autre hérite de la classe *Object*. La classe *String* hérite de la classe *Object*. La classe *Object* n'hérite d'aucune classe. On dit qu'un type τ est bien formé, et on note τ *bf*, s'il est soit `boolean`, soit `int`, soit *Object*, soit *String*, soit une classe C déclarée dans le fichier source.

La relation de sous-typage $\tau_1 \sqsubseteq \tau_2$ signifie « le type τ_1 est un sous-type du type τ_2 » et est définie par les règles d'inférence suivantes :

$$\frac{\tau \in \{\text{boolean}, \text{int}\}}{\tau \sqsubseteq \tau} \quad \frac{C_1 \longrightarrow C_2}{C_1 \sqsubseteq C_2} \quad \frac{}{\text{typenull} \sqsubseteq C}$$

Intuitivement, on peut interpréter la relation $\tau_1 \sqsubseteq \tau_2$ par « toute valeur de type τ_1 peut être donnée là où est attendue une valeur de type τ_2 ». On dit que les types τ_1 et τ_2 sont *compatibles*, et on note $\tau_1 \equiv \tau_2$, si $\tau_1 \sqsubseteq \tau_2$ ou $\tau_2 \sqsubseteq \tau_1$.

La notion de sous-typage est étendue aux profils (listes de types) de la manière suivante :

$$(\tau_1, \dots, \tau_n) \sqsubseteq (\tau'_1, \dots, \tau'_n) \text{ si et seulement si } \tau_i \sqsubseteq \tau'_i \text{ pour tout } i \in 1, \dots, n.$$

2.2 Champs, constructeurs et méthodes d'une classe

Dans toute la suite, C_0 désigne la classe courante, c'est-à-dire la classe que l'on est en train de typer.

On note $C\{ \tau \ x \}$ le fait que la classe C possède un champ x de type τ . On a $C\{ \tau \ x \}$ si et seulement si :

- la classe C déclare un champ x de type τ , ou
- la classe C est une sous-classe de C' et C' possède un champ x de type τ .

On note $C\{ C(\tau_1, \dots, \tau_n) \}$ le fait que l'ensemble des constructeurs de la classe C de profils plus grands que (τ_1, \dots, τ_n) pour la relation \sqsubseteq est non vide et admet un plus petit élément pour cette relation.

Soit C une classe, m un nom de méthode et τ_1, \dots, τ_n un profil. On définit l'ensemble $\text{meth}(C, m, \tau_1, \dots, \tau_n)$ comme l'union des $(n + 1)$ -uplets

- (C, a_1, \dots, a_n) pour toute méthode m de C de profil $(a_1, \dots, a_n) \sqsupseteq (\tau_1, \dots, \tau_n)$; et
- (C', a_1, \dots, a_n) pour toute méthode m d'une sur-classe C' de C de profil $(a_1, \dots, a_n) \sqsupseteq (\tau_1, \dots, \tau_n)$

On note alors $C\{ \tau m(\tau_1, \dots, \tau_n) \}$ le fait que l'ensemble $meth(C, m, \tau_1, \dots, \tau_n)$ est non vide et admet un unique plus petit élément pour \sqsubseteq , la méthode correspondante ayant pour type de retour τ .

2.3 Typage des expressions

Un contexte Γ est une suite de déclarations de variables de la forme $\tau_1 x_1, \dots, \tau_n x_n$. Il ne sera utilisé que pour les variables locales, les arguments de constructeurs et de méthodes, et éventuellement **this**. On introduit le jugement $\Gamma \vdash e : \tau$ signifiant « dans le contexte Γ , l'expression e est bien typée de type τ » et le jugement $\Gamma \vdash_l e : \tau$ signifiant « dans le contexte Γ , l'expression e est une valeur gauche bien typée de type τ ». Ces jugements sont alors définis par les règles d'inférence suivantes :

$$\begin{array}{c}
\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{}{\Gamma \vdash \text{null} : \text{typenull}} \qquad \frac{C \text{ this} \in \Gamma}{\Gamma \vdash \text{this} : C} \qquad \frac{\tau x \in \Gamma}{\Gamma \vdash_l x : \tau} \\
\\
\frac{x \notin \Gamma \quad C_0\{ \tau x \}}{\Gamma \vdash_l x : \tau} \qquad \frac{\Gamma \vdash e : C \quad C\{ \tau x \}}{\Gamma \vdash_l e.x : \tau} \\
\\
\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash e_1 = e_2 : \tau_1} \\
\\
\frac{\Gamma \vdash_l e : \text{int}}{\Gamma \vdash ++e : \text{int}} \qquad \frac{\Gamma \vdash_l e : \text{int}}{\Gamma \vdash --e : \text{int}} \qquad \frac{\Gamma \vdash_l e : \text{int}}{\Gamma \vdash e++ : \text{int}} \qquad \frac{\Gamma \vdash_l e : \text{int}}{\Gamma \vdash e-- : \text{int}} \\
\\
\frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash -e : \text{int}} \qquad \frac{\Gamma \vdash e : \text{boolean}}{\Gamma \vdash !e : \text{boolean}} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2 \quad op \in \{=, !=\}}{\Gamma \vdash e_1 op e_2 : \text{boolean}} \\
\\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad op \in \{<, <=, >, >=\}}{\Gamma \vdash e_1 op e_2 : \text{boolean}} \\
\\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad op \in \{+, -, *, /, \%\}}{\Gamma \vdash e_1 op e_2 : \text{int}} \\
\\
\frac{\Gamma \vdash e_1 : \text{boolean} \quad \Gamma \vdash e_2 : \text{boolean} \quad op \in \{\&\&, ||\}}{\Gamma \vdash e_1 op e_2 : \text{boolean}} \\
\\
\frac{\Gamma \vdash e_1 : \text{String} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \in \{\text{int}, \text{String}\}}{\Gamma \vdash e_1 + e_2 : \text{String}} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \text{String} \quad \tau_1 \in \{\text{int}, \text{String}\}}{\Gamma \vdash e_1 + e_2 : \text{String}} \\
\\
\frac{}{\Gamma \vdash e : \text{String}} \\
\\
\frac{}{\Gamma \vdash \text{System.out.print}(e) : \text{void}} \\
\\
\frac{\Gamma \vdash e_i : \tau_i \quad C_0\{ \tau m(\tau_1, \dots, \tau_n) \}}{\Gamma \vdash m(e_1, \dots, e_n) : \tau}
\end{array}$$

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 : \textit{String} \quad \Gamma \vdash e_2 : \textit{String}}{\Gamma \vdash e_1.\textit{equals}(e_2) : \textit{boolean}} \\
\frac{\Gamma \vdash e : C \quad \Gamma \vdash e_i : \tau_i \quad C\{ \tau \ m(\tau_1, \dots, \tau_n) \}}{\Gamma \vdash e.m(e_1, \dots, e_n) : \tau} \\
\frac{\Gamma \vdash e_i : \tau_i \quad C\{ C(\tau_1, \dots, \tau_n) \}}{\Gamma \vdash \textit{new } C(e_1, \dots, e_n) : C} \\
\frac{\Gamma \vdash e : \tau' \quad \tau \equiv \tau'}{\Gamma \vdash (\tau)e : \tau} \quad \frac{\Gamma \vdash e : \tau' \quad \tau \equiv \tau' \quad \tau' \in \{C, \textit{typenull}\}}{\Gamma \vdash e \textit{instanceof } \tau : \textit{boolean}}
\end{array}$$

2.4 Typage des instructions

On introduit le jugement $\Gamma \vdash i \rightarrow \Gamma'$ signifiant « dans le contexte Γ , l’instruction i est bien typée et définit le nouveau contexte Γ' ». Ce jugement est établi par les règles d’inférence suivantes :

$$\begin{array}{c}
\frac{}{\Gamma \vdash ; \rightarrow \Gamma} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e; \rightarrow \Gamma} \quad \frac{x \notin \Gamma \quad \tau \textit{ bf}}{\Gamma \vdash \tau x; \rightarrow \Gamma, \tau x} \quad \frac{x \notin \Gamma \quad \tau \textit{ bf} \quad \Gamma \vdash e : \tau' \quad \tau' \sqsubseteq \tau}{\Gamma \vdash \tau x = e; \rightarrow \Gamma, \tau x} \\
\frac{\Gamma \vdash e : \textit{boolean} \quad \Gamma \vdash i_1 \rightarrow \Gamma_1 \quad \Gamma \vdash i_2 \rightarrow \Gamma_2}{\Gamma \vdash \textit{if } (e) \ i_1 \ \textit{else } \ i_2 \rightarrow \Gamma} \\
\frac{\Gamma \vdash e_1; \rightarrow \Gamma \quad \Gamma \vdash e_2 : \textit{boolean} \quad \Gamma \vdash e_3; \rightarrow \Gamma \quad \Gamma \vdash i \rightarrow \Gamma_1}{\Gamma \vdash \textit{for}(e_1; e_2; e_3) \ i \rightarrow \Gamma} \\
\frac{\Gamma \vdash i_1 \rightarrow \Gamma_1 \quad \Gamma_1 \vdash i_2 \rightarrow \Gamma_2}{\Gamma \vdash i_1; i_2 \rightarrow \Gamma_2} \\
\frac{\Gamma \vdash i \rightarrow \Gamma'}{\Gamma \vdash \{i\} \rightarrow \Gamma} \quad \frac{}{\Gamma \vdash \textit{return}; \rightarrow \Gamma} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textit{return } e; \rightarrow \Gamma}
\end{array}$$

De plus, on a les équivalences suivantes :

- $\textit{if } (e_1) \ e_2$ équivaut à $\textit{if } (e_1) \ e_2 \ \textit{else};$,
- si e_1 ou e_3 est omis dans $\textit{for}(e_1; e_2; e_3)$ alors il équivaut à $;$,
- si e_2 est omis dans $\textit{for}(e_1; e_2; e_3)$ alors il équivaut à \textit{true} .

2.5 Typage des classes

2.5.1 Conditions d’existence et d’unicité

Pour être sémantiquement correct, un fichier doit tout d’abord respecter les contraintes suivantes :

- chaque classe ne peut être définie qu’une seule fois ;
- une classe ne peut hériter d’une classe qui n’est pas définie, ni hériter de la classe *String* ;
- la relation d’héritage ne peut admettre de cycle.

Notez bien que l’ordre d’apparition des classes dans le fichier n’a aucune importance : en tout point, on peut référencer une classe définie ailleurs dans le fichier (avant ou après).

D’autre part, les conditions suivantes doivent être remplies :

- tous les champs d’une même classe doivent porter des noms différents ;

- tous les constructeurs d’une même classe doivent avoir des profils différents et porter le même nom que la classe.
- deux méthodes de même nom d’une même classe doivent avoir des *arguments* différant en nombre et/ou en types.

2.5.2 Redéfinition

Une méthode m est dite *redéfinie* si elle apparaît avec un même profil (arguments de mêmes types) dans une classe C et une sous-classe C' de C . Dans ce cas, ces deux méthodes doivent avoir le même type de retour.

2.5.3 Typage des champs, constructeurs et méthodes

Soit C_0 la classe courante. Le contexte initial de typage est $\Gamma_0 = C_0 \text{ this}$, ne contenant que la « variable » `this` de type C_0 .

Typage des champs. Pour un champ τx , on doit uniquement vérifier que le type τ est bien formé.

Typage des constructeurs. Un constructeur $C_0(\tau_1 x_1, \dots, \tau_n x_n)\{i\}$ est bien formé si tous les identificateurs x_i sont deux à deux distincts, si tous les types τ_i sont bien formés, et si la liste d’instructions i est bien typée dans le contexte $\Gamma_0, \tau_1 x_1, \dots, \tau_n x_n$.

Typage des méthodes. Une méthode $\tau m(\tau_1 x_1, \dots, \tau_n x_n)\{i\}$ est bien formée si tous les identificateurs x_i sont deux à deux distincts, si tous les types τ_i sont bien formés, et si la liste d’instructions i est bien typée dans le contexte $\Gamma_0, \tau_1 x_1, \dots, \tau_n x_n$.

D’autre part, toute occurrence de l’instruction `return` dans i doit retourner une valeur d’un type sous-type de τ . Enfin, lorsque le type τ n’est pas `void`, tout flot d’exécution possible dans i doit contenir (au moins) une instruction `return`.

Remarque : à la différence du compilateur Java, on ne cherchera pas à signaler le code inatteignable (*i.e.*, derrière un `return`).

2.6 Indications

Comment procéder. On suggère fortement de réaliser l’analyse sémantique en trois phases distinctes, à savoir :

1. déclaration des classes et vérification de leur unicité,
2. construction de la relation d’héritage, déclaration des champs, constructeurs et méthodes de toutes les classes (avec les vérifications adéquates),
3. typage des corps des constructeurs et des méthodes.

Vous restez cependant libres d’utiliser toute autre méthode de votre choix qui se révélerait correcte.

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

Anticipation. Dans la phase suivante (production de code), certaines informations provenant du typage seront nécessaires, telles que par exemple la nécessité de transformer une valeur entière en une chaîne de caractères, 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 Limitations par rapport à Java

Le langage Petit Java souffre d'un certain nombre de limitations par rapport à Java, à savoir :

- `System.out.print` n'est pas ici surchargée pour accepter d'autres types que *String* en argument. Pour afficher un entier *i* on écrira donc `System.out.print(i + "")`.
- La classe *String* ne possède que la méthode `equals`.
- La classe *Object* ne possède aucune méthode.
- Les classes n'ont pas de constructeur par défaut.
- Les arguments de la ligne de commande ne sont pas utilisables dans le programme (*i.e.*, l'argument de la méthode `main` de la classe `Main` n'est pas visible, car il n'y a pas de tableaux dans Petit Java).

D'autre part, Petit Java possède moins de mots clés que Java. Cependant, votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Petit Java mais corrects au sens de Java.

4 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 MIPS. On ne cherchera pas à libérer la mémoire.

4.1 Organisation des données

4.1.1 Représentation des valeurs

Types primitifs. Les entiers seront représentés de manière immédiate par les entiers MIPS (32 bits signés). Les booléens sont représentés par des entiers, avec la même convention que MIPS : 0 représente `false` et une valeur non nulle représente `true`.

Objets. Un objet est représenté par une adresse, pointant sur un bloc alloué sur le tas. Ce bloc contient

- un *descripteur de classe* (voir ci-dessous) ;
- les valeurs des champs de l'objet.

Les champs sont organisés de telle manière qu'un même champ d'une classe C se trouve dans le bloc à la même position pour tout objet de la classe C ou de toute sous-classe de C (organisation dite en *préfixe*). Ceci est possible car l'héritage est *simple*. Ainsi, les deux classes A et B suivantes :

```
class A { int x; boolean b; }
class B extends A { int d; }
```

donneront lieu à des objets ayant la forme suivante :

descr. A	descr. B
x	x
b	b
	d

Le compilateur maintiendra 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 .

La valeur null. Elle est représentée par une adresse sur le tas, différente de toute autre adresse d'objet. Le plus simple est de l'allouer dès le démarrage du programme, dans le segment de données.

Les classes prédéfinies *Objet* et *String*. Les objets de ces classes seront représentés exactement comme les autres, avec des descripteurs particuliers pour ces deux classes. Un objet de la classe *String* possédera en outre un champ contenant un pointeur vers la chaîne qu'il représente (terminée par zéro).

4.1.2 Descripteurs de classes

Chaque classe est représentée par un descripteur de classe. Il sera par exemple alloué dans le segment de données. Il contient :

- le descripteur de sa super-classe ;
- la liste des codes des méthodes.

Exactement comme pour la représentation des objets, la liste de ces codes obéit à une organisation préfixe : le code pour la méthode **f** doit se trouver au même endroit dans le descripteur de la classe A qui la définit et dans celui de la classe B qui la redéfinit. Ainsi le code suivant :

```
class A {
    int f() { return 0; }
    boolean g() { return true; }
}
class B extends A {
    int f() { return 1; }
    boolean h() { return false; }
}
```

donnera lieu à des descripteurs de la forme suivante :

descr. de A :	descr. de Object
	A_f
	A_g

descr. de B :	descr. de A
	B_f
	A_g
	B_h

De même que pour les champs, le compilateur maintiendra une table donnant, pour chaque classe C et chaque méthode de C , la position où trouver le code de cette méthode dans le descripteur de la classe C .

4.2 Schéma de compilation

4.2.1 Méthodes et constructeurs

Les méthodes et les constructeurs sont représentés par des fonctions (*i.e.*, du code appelé par un `jal`). L'appelant place la valeur de l'objet (`this`) et les arguments sur la pile. L'appelé place la valeur de retour dans le registre `$v0`. Au retour, l'appelant se charge de dépiler `this` et les arguments.

4.2.2 Transtypage et `instanceof`

Les opérations de transtypage (`cast`) et `instanceof`, lorsqu'elles ne peuvent être résolues statiquement (au typage), doivent être effectuées dynamiquement. Pour cela on exploite le fait que les objets contiennent des descripteurs de classes, contenant eux-mêmes les descripteurs de leur super-classe. Ainsi on peut faire la vérification adéquate en suivant ce chaînage depuis la classe de l'objet jusqu'à la classe désirée. En cas de succès, le transtypage n'a pas d'effet (laissant l'objet inchangé) et `instanceof` renvoie `true`. Si l'on parvient à la classe `Object` sans succès, alors le transtypage échoue (par exemple avec le message "`cast failure`") et `instanceof` renvoie `false`. Le plus simple est d'écrire deux fonctions MIPS pour ces deux opérations.

Notez qu'il convient de tester si l'objet est `null` avant de démarrer cette vérification.

4.2.3 Ordre des opérations de compilation

On pourra effectuer les différentes opérations nécessaires à la production de code dans l'ordre suivant :

1. Construction des descripteurs de classe et de la table décrivant les objets (indiquant où se trouvent les champs au sein des objets).

Cette étape nécessite que les classes soient traitées dans un ordre compatible avec l'héritage (les classes les plus vieilles étant traitées les premières). Pour cela il suffit de suivre l'algorithme suivant

```

traiter(C) ≡ si C a déjà été traitée, ne rien faire ;
             sinon,
             si C hérite de C', appeler traiter(C')
             construire le descripteur de C
             remplir la table des champs de C

```

et d'appeler `traiter(C)` sur toutes les classes C du fichier. L'absence de cycle dans la relation d'héritage, vérifiée au typage, garantit la terminaison de cet algorithme.

2. Compilation de toutes les fonctions (code des constructeurs et des méthodes, ainsi que des fonctions réalisant les opérations de transtypage et `instanceof`).

Remarque importante. La correction du projet sera réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec l'instruction `System.out.print`, qui seront compilés avec votre compilateur et dont la sortie sera comparée à la sortie attendue. Il est donc très important de correctement compiler les appels à `System.out.print`.

5 Travail demandé

Écrire un compilateur, appelons-le `pjava`, acceptant sur sa ligne de commande exactement un fichier `Petit Java` (portant l'extension `.java`) et éventuellement l'option `-parse-only` ou l'option `-type-only`. Ces deux options indiquent respectivement de stopper la compilation après l'analyse syntaxique (section 1) et l'analyse sémantique (section 2).

Si le fichier est conforme à la syntaxe et au typage décrits dans ce document, votre compilateur doit produire du code MIPS dans un fichier (portant le même nom que le fichier source mais avec le suffixe `.s`) et terminer avec le code de sortie 0 (`exit 0` explicite ou terminaison normale du programme), sans rien afficher.

En cas d'erreur lexicale, syntaxique ou de typage, celle-ci doit être signalée (de la manière indiquée ci-dessous) et le programme doit terminer avec le code de sortie 1 (`exit 1`). En cas d'autre erreur (une erreur du compilateur lui-même), le programme doit terminer avec le code de sortie 2 (`exit 2`).

Localisation des erreurs. Lorsqu'une erreur est détectée par votre compilateur, elle 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.java", 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.)

Les localisations peuvent être obtenues pendant l'analyse syntaxique grâce aux mots-clés `$startpos` et `$endpos` de Menhir, puis conservées dans l'arbre de syntaxe abstraite.

Modalités de remise de votre projet. 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 programme (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur,

qui sera appelé `pjava`. 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.

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`. La date limite de remise sera annoncée sur le site du cours.