projet de compilation

Petit Pyret

version 1 — 22 octobre 2025

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Pyret[1], appelé Petit Pyret par la suite, produisant du code x86-64. Il s'agit d'un fragment 100% compatible avec Pyret, au sens où tout programme de Petit Pyret est aussi un programme Pyret correct. On se donne notamment comme objectif de réaliser un typage statique de Petit Pyret. Le présent sujet décrit précisément Petit Pyret, 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)
$\langle règle \rangle_t^\star$	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, tabulations et retours chariot constituent des blancs. Les commentaires peuvent prendre deux formes :

- débutant par `#` et s'étendant jusqu'à la fin de la ligne ;
- débutant par `#|`, s'étendant jusqu'à `|#` et pouvant être imbriqués.

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

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

On note qu'un identificateur ne peut pas se terminer par un tiret (`-`). Les identificateurs suivants sont des mots clés :

```
and     block   cases   else    end     false   for
from    fun     if      lam     or      true    var
```

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

$$\langle integer \rangle \quad ::= \quad (\texttt{-} \mid \texttt{+})?\ \langle digit \rangle^+$$

1. https://pyret.org/

Une chaîne de caractères ⟨*string*⟩ s'écrit entre guillemets (`"`) ou entre apostrophes (`'`). Il y a cinq séquences d'échappement : `\"` pour le caractère `"`, `\'` 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 Syntaxe

La grammaire des fichiers sources est donnée figure 1. Le point d'entrée est le non terminal ⟨*file*⟩. À cette grammaire s'ajoutent un certain nombre de restrictions et de sucres syntaxiques, décrits ci-dessous. De nombreux tests, positifs et négatifs, sont fournis avec ce sujet pour aider à comprendre ce qui est attendu. La réalisation des contraintes ci-dessous peut être réalisée dans l'analyseur lexical, dans l'analyseur syntaxique ou conjointement dans les deux.

Blancs. Le langage Pyret est plutôt "tatillon" en ce qui concerne l'utilisation des blancs. Ainsi, tout opérateur binaire doit être précédé et suivi de blanc (espace, tabulation ou retour chariot). Inversement, il ne doit pas y avoir de blanc entre une fonction et la parenthèse ouvrante correspondant aux paramètres de l'appel, que ce soit les paramètres formels ou effectifs. Ainsi, on ne peut pas écrire `f (1)`, ni `f(1) (2)` (pour une fonction `f` qui renvoie une fonction dans ce second cas). De même, il ne doit pas y avoir de blanc entre `lam` et la parenthèse ouvrante correspondant aux paramètres formels. Dans une définition de fonction polymorphe, l'espace est interdit entre le caractère `>` et la parenthèse ouvrante correspondant aux paramètres formels. Enfin, il ne doit pas y avoir d'espace entre `block` et `:`, ni entre `else` et `:`. Indication : Il est fortement déconseillé de produire un lexème `WHITE` pour les blancs à l'analyse lexicale pour le traiter ensuite à l'analyse syntaxique. On cherchera plutôt une solution au niveau de l'analyse lexicale.

Priorité des opérateurs. Contrairement à beaucoup de langages, les opérateurs binaires de Pyret n'ont pas de priorités associées. Toute ambiguïté doit être levée par l'utilisateur à l'aide de parenthèses. En conséquence, toute expression enchaînant des opérations binaires est limitée à une seule et même opération. Ainsi, on peut écrire `1 - 2 - 3` mais pas `1 - 2 + 3`. Bien entendu, on peut utiliser des parenthèses, comme dans `1 + (2 - 3 - 4) + 5`. Par ailleurs, il n'y a pas d'opérateur unaire dans Petit Pyret.

Blocs. Un bloc correspondant au non terminal ⟨*block*⟩ est introduit avec `:` ou bien avec `block:` (c'est le rôle du non terminal ⟨*ublock*⟩). Un bloc doit toujours être terminé par une expression ou une affectation. Il ne peut y avoir plusieurs expressions ou affectations dans un bloc que si celui-ci a été introduit avec `block:`. En revanche, il peut y avoir autant de déclarations de variables et de fonctions qu'on le souhaite. À noter que la nature du bloc s'applique au corps d'une fonction, à tous les blocs d'une *même expression* `if`, et à toutes les branches d'une expression `cases`. Mais elle ne se propage pas en profondeur. Ainsi, une expression `if` dans le corps d'une fonction n'hérite pas du décorateur `block:` de la fonction.

Conditionnelle. Une expression conditionnelle (`if`) doit contenir au moins une branche `else:` ou une branche `else if`. En revanche, il peut n'y avoir que des `else if` sans `else:` final. Dans ce cas, l'absence de branche `else` correspond à `else: raise("error")`, où `raise` est une fonction prédéfinie (voir section suivante).

$$
\begin{array}{lcl}
\langle file \rangle & ::= & \langle stmt \rangle^\star \texttt{ EOF} \\
\langle block \rangle & ::= & \langle stmt \rangle^+ \\
\langle stmt \rangle & ::= & \texttt{fun } \langle ident \rangle \texttt{ (<} \langle ident \rangle^+_{\texttt{,}} \texttt{>)?} \langle funbody \rangle \\
& | & \texttt{var?} \langle ident \rangle \texttt{ (::} \langle type \rangle \texttt{)?} \texttt{ =} \langle bexpr \rangle \\
& | & \langle ident \rangle \texttt{ :=} \langle bexpr \rangle \\
& | & \langle bexpr \rangle \\
\langle funbody \rangle & ::= & \texttt{(} \langle param \rangle^\star_{\texttt{,}} \texttt{)} \langle rtype \rangle \langle ublock \rangle \langle block \rangle \texttt{ end} \\
\langle param \rangle & ::= & \langle ident \rangle \texttt{ ::} \langle type \rangle \\
\langle rtype \rangle & ::= & \texttt{->} \langle type \rangle \\
\langle ublock \rangle & ::= & \texttt{:} \\
& | & \texttt{block:} \\
\langle type \rangle & ::= & \langle ident \rangle \texttt{ (<} \langle type \rangle^+_{\texttt{,}} \texttt{>)?} \\
& | & \texttt{(} \langle type \rangle^\star_{\texttt{,}} \langle rtype \rangle \texttt{)} \\
\langle bexpr \rangle & ::= & \langle expr \rangle \ (\langle binop \rangle \ \langle expr \rangle)^\star \\
\langle expr \rangle & ::= & \texttt{true} \mid \texttt{false} \mid \langle integer \rangle \mid \langle string \rangle \\
& | & \langle ident \rangle \\
& | & \texttt{(} \langle bexpr \rangle \texttt{)} \\
& | & \texttt{block:} \langle block \rangle \texttt{ end} \\
& | & \texttt{if } \langle bexpr \rangle \langle ublock \rangle \langle block \rangle \texttt{ (else if } \langle bexpr \rangle \texttt{ :} \langle block \rangle)^\star \\
& & \texttt{(else:} \langle block \rangle)? \texttt{ end} \\
& | & \langle caller \rangle \texttt{ (} \langle bexpr \rangle^\star_{\texttt{,}} \texttt{)} \\
& | & \texttt{lam } \langle funbody \rangle \\
& | & \texttt{cases (} \langle type \rangle \texttt{)} \langle bexpr \rangle \langle ublock \rangle \langle branch \rangle^\star \texttt{ end} \\
& | & \texttt{for } \langle caller \rangle \texttt{ (} \langle from \rangle^\star_{\texttt{,}} \texttt{)} \langle rtype \rangle \langle ublock \rangle \langle block \rangle \texttt{ end} \\
\langle caller \rangle & ::= & \langle ident \rangle \\
& | & \langle caller \rangle \texttt{ (} \langle bexpr \rangle^\star_{\texttt{,}} \texttt{)} \\
\langle branch \rangle & ::= & \texttt{|} \langle ident \rangle \texttt{ ((} \langle ident \rangle^\star_{\texttt{,}} \texttt{)?} \texttt{=>} \langle block \rangle \\
\langle from \rangle & ::= & \langle param \rangle \texttt{ from } \langle bexpr \rangle \\
\langle binop \rangle & ::= & \texttt{==} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{and} \mid \texttt{or}
\end{array}
$$

FIGURE 1 – Grammaire de Petit Pyret.

Boucle `for`. L'expression

$$\begin{array}{l}\texttt{for } c\texttt{(} x_1\texttt{::}\tau_1 \texttt{ from } e_1 \texttt{ , ... , } x_n\texttt{::}\tau_n \texttt{ from } e_n \texttt{) -> } \tau \texttt{ :} \\ \quad b \\ \texttt{end}\end{array}$$

se désucre en

$$c\ \texttt{(lam(}\ x_1\texttt{::}\tau_1\ \texttt{, ... ,}\ x_n\texttt{::}\tau_n\ \texttt{) -> }\tau\ \texttt{:}\ b\ \texttt{end,}\ e_1\ \texttt{, ... ,}\ e_n\ \texttt{)}$$

Ceci permet notamment d'utiliser des fonctions prédéfinies comme `each` ou `fold` pour parcourir des listes (voir dans la section suivante pour le type de ces deux fonctions).

2 Typage statique

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

$$\begin{array}{rcl}\tau & ::= & \alpha \mid \texttt{Any} \mid \texttt{Nothing} \mid \texttt{Boolean} \mid \texttt{Number} \mid \texttt{String} \mid \texttt{List}\langle\tau\rangle \mid (\tau, \ldots, \tau \to \tau) \\ \sigma & ::= & \forall\alpha, \ldots, \alpha.\, \tau\end{array}$$

Les types sont munis d'une relation de sous-typage, notée $\tau_1 \leq \tau_2$ et définie par les règles suivantes :

$$\frac{}{\tau \leq \texttt{Any}} \qquad \frac{}{\tau \leq \tau} \qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \qquad \frac{\tau_1 \leq \tau_2}{\texttt{List}\langle\tau_1\rangle \leq \texttt{List}\langle\tau_2\rangle}$$

$$\frac{\tau \leq \tau' \quad \forall i.\, \tau'_i \leq \tau_i}{(\tau_1, \ldots, \tau_n \to \tau) \leq (\tau'_1, \ldots, \tau'_n \to \tau')}$$

Deux types sont dits équivalents, noté $\tau_1 \equiv \tau_2$, si $\tau_1 \leq \tau_2$ et $\tau_2 \leq \tau_1$.

Un environnement de typage, noté Γ, contient un ensemble de variables de types $\alpha_1, \ldots, \alpha_n$ et un ensemble de déclarations de variables de la forme $x : \sigma$ pour une variable immuable ou $\texttt{var}\ x : \tau$ pour une variable mutable. Le jugement $\Gamma \vdash \tau\ bf$ signifie « le type τ est bien formé dans l'environnement Γ », c'est-à-dire que toutes les variables de types apparaissant dans τ sont bien présentes dans Γ.

Initialement, l'environnement ne contient aucune variable de type et contient les variables (immuables) suivantes :

$$\begin{array}{rcl}\texttt{nothing} & : & \texttt{Nothing} \\ \texttt{num-modulo} & : & (\texttt{Number}, \texttt{Number} \to \texttt{Number}) \\ \texttt{empty} & : & \forall\alpha.\, \texttt{List}\langle\alpha\rangle \\ \texttt{link} & : & \forall\alpha.\, (\alpha, \texttt{List}\langle\alpha\rangle \to \texttt{List}\langle\alpha\rangle) \\ \texttt{print} & : & \forall\alpha.\, (\alpha \to \alpha) \\ \texttt{raise} & : & \forall\alpha.\, (\texttt{String} \to \alpha) \\ \texttt{each} & : & \forall\alpha\,\beta.\, ((\alpha \to \beta), \texttt{List}\langle\alpha\rangle \to \texttt{Nothing}) \\ \texttt{fold} & : & \forall\alpha\,\beta.\, ((\alpha, \beta \to \alpha), \alpha, \texttt{List}\langle\beta\rangle \to \alpha)\end{array}$$

On note $\Gamma{+}\alpha$ l'environnement Γ étendu avec une nouvelle variable de type α, sous l'hypothèse que α n'apparaisse pas déjà dans Γ et ne porte pas le nom d'un type prédéfini. De même, $\Gamma{+}x : \sigma$ désigne l'environnement Γ étendu avec une nouvelle déclaration de variable, sous l'hypothèse que x n'apparaît pas déjà dans Γ ; même chose avec $\Gamma + \texttt{var}\ x : \tau$. En particulier, il n'est pas possible d'introduire une variable qui cacherait un symbole prédéfini (comme `empty` ou `print`),

ni une variable locale qui cacherait une variable globale ou une variable locale précédemment introduite.

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 d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\texttt{var}\ x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{x : \forall \alpha_1, \ldots, \alpha_n.\, \tau \in \Gamma \quad \forall i,\, \Gamma \vdash \tau_i\ bf}{\Gamma \vdash x : \tau[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]}$$

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

$$\frac{\Gamma \vdash e_1 : \texttt{String} \quad \Gamma \vdash e_2 : \texttt{String}}{\Gamma \vdash e_1\ \texttt{+}\ e_2 : \texttt{String}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad op \in \{\texttt{==},\ \texttt{<>}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Boolean}}$$

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

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

$$\frac{\Gamma \vdash e : \texttt{Boolean} \quad \Gamma \vdash b_1 : \tau_1 \quad \Gamma \vdash b_2 : \tau_2 \quad \tau_1 \equiv \tau_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{:}\ b_1\ \texttt{else}\ b_2 : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau' \quad \tau' \leq \texttt{List}\langle\tau\rangle \quad \Gamma \vdash b_1 : \tau_1 \quad \Gamma + x : \tau + y : \texttt{List}\langle\tau\rangle \vdash b_2 : \tau_2 \quad \tau_1 \equiv \tau_2}{\Gamma \vdash \texttt{cases(List}\langle\tau\rangle\texttt{)}\ e\ \texttt{:}\ \texttt{|}\ \texttt{empty =>}\ b_1\ \texttt{|}\ \texttt{link(}x, y\texttt{) =>}\ b_2\ \texttt{end} : \tau_1}$$

$$\frac{(\forall i, \Gamma \vdash \tau_i\ bf) \quad \Gamma + x_1 : \tau_1 + \cdots + x_n : \tau_n \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash \texttt{lam(}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{)->}\tau\texttt{:}\ e\ \texttt{end} : (\tau_1, \ldots, \tau_n \to \tau)}$$

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n \to \tau) \quad \forall i,\ \Gamma \vdash e_i : \tau_i' \quad \tau_i' \leq \tau_i}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{block:}\ e\ \texttt{end} : \tau} \qquad \frac{\texttt{var}\ x : \tau \in \Gamma \quad \Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash \texttt{block:}\ x\ \texttt{:=}\ e\ \texttt{end} : \texttt{Nothing}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \texttt{block:}\ s \cdots\ \texttt{end} : \tau_2}{\Gamma \vdash \texttt{block:}\ e_1\texttt{;}\ s \cdots\ \texttt{end} : \tau_2}$$

$$\frac{\texttt{var}\ x : \tau_1 \in \Gamma \quad \Gamma \vdash e_1 : \tau_1' \quad \tau_1' \leq \tau_1 \quad \Gamma \vdash \texttt{block:}\ s \cdots\ \texttt{end} : \tau_2}{\Gamma \vdash \texttt{block:}\ x\ \texttt{:=}\ e_1\texttt{;}\ s \cdots\ \texttt{end} : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \mathit{Gen}(\Gamma, \tau_1) \vdash \texttt{block:}\ s \cdots\ \texttt{end} : \tau_2}{\Gamma \vdash \texttt{block:}\ x\ \texttt{=}\ e_1\texttt{;}\ s \cdots\ \texttt{end} : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1' \quad \tau_1' \leq \tau_1 \quad \Gamma + x : \mathit{Gen}(\Gamma, \tau_1) \vdash \texttt{block:}\ s \cdots\ \texttt{end} : \tau_2}{\Gamma \vdash \texttt{block:}\ x\texttt{::}\ \tau_1\ \texttt{=}\ e_1\texttt{;}\ s \cdots\ \texttt{end} : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + \texttt{var}\ x : \tau_1 \vdash \texttt{block:}\ s \cdots\ \texttt{end} : \tau_2}{\Gamma \vdash \texttt{block:}\ \texttt{var}\ x\ \texttt{=}\ e_1\texttt{;}\ s \cdots\ \texttt{end} : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1' \quad \tau_1' \leq \tau_1 \quad \Gamma + \texttt{var}\ x : \tau_1 \vdash \texttt{block:}\ s \cdots\ \texttt{end} : \tau_2}{\Gamma \vdash \texttt{block:}\ \texttt{var}\ x\texttt{::}\ \tau_1\ \texttt{=}\ e_1\texttt{;}\ s \cdots\ \texttt{end} : \tau_2}$$

$$\frac{\begin{array}{c}\Gamma' \stackrel{\text{def}}{=} \Gamma + \alpha_1 + \cdots + \alpha_m \quad (\forall i, \Gamma' \vdash \tau_i\ bf) \quad \sigma \stackrel{\text{def}}{=} \forall \alpha_1, \ldots, \alpha_m.\, (\tau_1, \ldots, \tau_n \to \tau) \\ \Gamma' + f : \sigma + x_1 : \tau_1 + \cdots + x_n : \tau_n \vdash e : \tau' \quad \tau' \leq \tau \\ \Gamma' + f : \sigma \vdash \texttt{block:}\ s \cdots\ \texttt{end} : \tau_2\end{array}}{\Gamma \vdash \texttt{block: fun}\ f\texttt{<}\alpha_1, \ldots, \alpha_m\texttt{>(}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{)->}\tau\texttt{:}\ e\ \texttt{end}\ s \cdots\ \texttt{end} : \tau_2}$$

Dans les règles ci-dessus, la notation $Gen(\Gamma, \tau)$ désigne le schéma $\forall \alpha_1, \ldots, \alpha_n.\, \tau$ où les α_i sont les variables de type apparaissant dans τ mais pas dans Γ. Dans la construction `cases`, les deux branches peuvent être interverties (le cas `link` avant le cas `empty`) et par ailleurs les variables x et y du motif `link` peuvent être ignorées en remplaçant l'une et/ou l'autre par `_` (auquel cas, elles ne sont pas ajoutées à l'environnement). Un fichier est typé comme un bloc.

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.

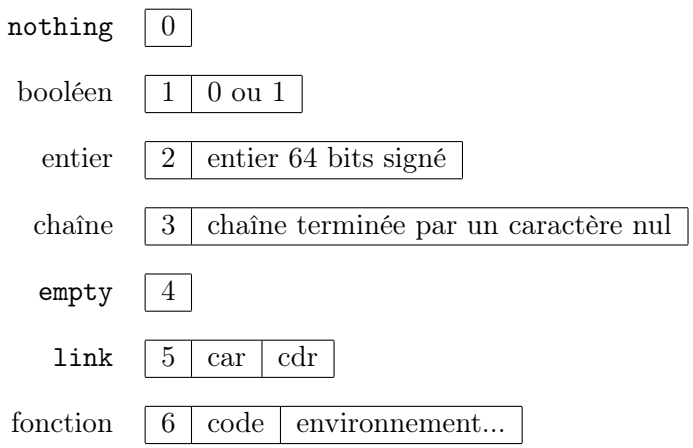
En cas de doute concernant un point de sémantique, vous pouvez utiliser le compilateur Pyret (ou sa version en ligne) comme référence. Vous pouvez d'ailleurs vous inspirer de ses messages d'erreur pour votre compilateur (en les traduisant ou non en français). Mais attention, le typeur statique de Pyret (option `-y` du compilateur Pyret / bouton "Type-check dans Run" dans la version en ligne) est encore très expérimental, avec des limitations et des bugs. Notre objectif ici est de faire mieux !

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.

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.

Représentation des valeurs. On suggère un schéma simple où toute valeur est une adresse vers un bloc alloué sur le tas, dont le premier mot est une étiquette entière indiquant sa nature et dont les octets suivants contiennent la valeur, le cas échéant.



Note : Les entiers de Pyret (type `Number`) sont en réalité des rationnels de précision arbitraire, mais on opte ici pour de simples entiers 64 bits signés. On pourra bien entendu pré-allouer `nothing`, les booléens et `empty`.

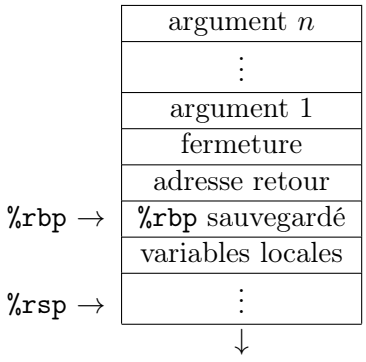


Schéma de compilation. Chaque fonction de Petit Pyret est compilée vers une fonction assembleur qui prend tous ses arguments sur la pile et place sa valeur de retour dans le registre `%rax`. Si la fonction est une fermeture, celle-ci est passée comme un tout premier paramètre. Cela simplifie les choses 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 primitives `each`/`fold` qui sont en Pyret 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, `print`, 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 via Moodle [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 `pyretc`. 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.

2. https://moodle.psl.eu/course/view.php?id=34905

Partie 1 (à rendre pour le dimanche 14 décembre 18:00). Dans cette première partie du projet, le compilateur `pyretc` doit accepter sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Petit Pyret portant l'extension `.arr`. Il doit alors réaliser l'analyse syntaxique du fichier. En cas d'erreur lexicale ou syntaxique, celle-ci doit être signalée par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

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

L'anglicisme de la première ligne est imposé mais 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 (par exemple, `exit 1` en OCaml).

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 de la manière suivante :

```
File "test.arr", line 4, characters 5-6:
this expression has type Number but is expected to have type String
```

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. 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. 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 18 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.arr`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.arr`). 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 :

```
pyret -q -e none file.arr
```

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. (Voir les tests fournis dans le sous-répertoire `exec/`.) 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 (`print`), arithmétique, variables, `if`, fonctions, filtrage, fonctions de première classe.