# Projet : Analyseurs syntaxique et sémantique pour un sous-langage de Rust

**Assembleur – Compilation, ENSIIE**

Semestre 3, 2023–24

## 1 Informations pratiques

Le code rendu comportera un Makefile, et devra pouvoir être compilé avec la commande `make`. **Tout projet ne compilant pas se verra attribuer un 0** : mieux vaut rendre un code incomplet mais qui compile, qu'un code ne compilant pas. Votre code devra être **abondamment commenté et documenté**.

L'analyseur syntaxique demandé à la question 2 sera impérativement obtenu avec les outils lex/yacc [1] ou ocamllex/ocamlyacc [2]. Ceci implique donc que votre projet sera écrit au choix en C ou en OCaml.

Des fichiers d'entrée pour tester votre code seront disponibles dans une archive à l'adresse : http://www.ensiie.fr/~guillaume.burel/compilation/projet_rustine.tar.gz . Vous attacherez un soin particulier à ce que ces exemples fonctionnent. (Il y a à la fois des exemples d'entrées correctes et d'entrées que le compilateur est censé rejeter.)

Votre projet est à déposer sur http://exam.ensiie.fr dans le dépôt `asscom_proj_2023` sur forme d'une archive `tar.gz` **avant le 7 janvier 2024** inclus. **Tout projet rendu en retard se verra attribuer la note 0.** Vous n'oublierez pas d'inclure dans votre dépôt un rapport (PDF) précisant vos choix, les problèmes techniques qui se posent et les solutions trouvées.

Toute tentative de fraude (plagiat, etc.) sera sanctionnée. Si plusieurs projets ont **des sources trop similaires** (y compris sur une partie du code uniquement), *tous* leurs auteurs se verront attribuer la note 0/20.

## 2 Sujet

Rust est un langage qui permet d'avoir un contrôle de bas niveau, tout en ayant une gestion sûre de la mémoire sans avoir de surcoût (par exemple celui lié à la présence d'un *garbage collector*, inutile en Rust).

Le but de ce projet est d'écrire un exécutable qui prend en entrée un programme écrit en Rustine, un sous-langage de Rust, qui construit son arbre de syntaxe abstraite et qui fait quelques analyses sémantiques dessus. La syntaxe de Rustine est décrite par ce qui suit :

1. Documentation disponible à la page http://dinosaur.compilertools.net/. On peut aussi utiliser les versions libres flex/bison.
2. " http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html

Un commentaire commence soit par `//` et se termine à la fin de la ligne, soit par `/*` et se termine par `*/`. Ce deuxième type de commentaires peut être imbriqué (donc comme en OCaml et pas comme en C).

Les mots-clefs en Rustine sont `as break continue else false fn if let loop mut return true while` ; les mots réservés sont `abstract async await become box const crate do dyn enum extern final for impl in macro match mod move override priv pub ref self Self static struct super trait type typeof unsafe unsized use virtual where yield`.

Les identifiants en Rustine ne peuvent pas être des mots clefs ou des mots réservés. Ils commencent par une lettre (minuscule ou majuscule) ou un underscore, suivi d'un nombre potentiellement nul de lettres, de chiffres et d'underscore. Il est également possible d'utiliser les caractères `r#` suivis d'une lettre (minuscule ou majuscule) ou un underscore, puis d'un nombre potentiellement nul de lettres, de chiffres et d'underscore. Dans ce cas, l'identifiant est ce qui suit `r#`, et il peut être égal à un mot-clé ou un mot réservé. Ainsi, `a_B1` et `r#a_B1` représentent le même identifiant ; `box` n'est pas un identifiant ; `r#while` est l'**identifiant** `while`, à ne pas confondre avec le mot-clé `while`.

Une étiquette est `'` suivi d'une lettre (minuscule ou majuscule) ou un underscore, suivi d'un nombre potentiellement nul de lettres, de chiffres et d'underscore, par exemple `'label`, `'_` ou `'return`.

En Rustine, on ne considère que les types numériques (entiers et nombres à virgule flottante) de Rust. Les types de bases seront donc `i8 i16 i32 i64 i128 isize` pour les entiers signés d'une taille (en bits) donnée ; `u8 u16 u32 u64 u128 usize` pour les entiers non-signés d'une taille (en bits) donnée ; `f32 f64` pour les nombres à virgule flottante simple et double précision ; `bool` pour les booléens ; et `()` pour le type unit (les parenthèses peuvent être séparées par des espaces, de retours à la ligne ou des commentaires). Il est possible d'obtenir un type en préfixant un autre type par `&` (référence) ou par `&` suivi de `mut` (référence mutable). Il est également possible de mettre des parenthèses non significatives autour d'un type.

Les constantes entières seront :

— des constantes décimales : une suite non vide de chiffres entre `0` et `9` et d'underscores qui ne commence pas par un underscore ;

— des constantes binaires : `0b` suivi d'une suite non vide de chiffres `0` ou `1` et d'underscores qui contient au moins un chiffre ;

— des constantes octales : `0o` suivi d'une suite non vide de chiffres entre `0` et `7` et d'underscores qui contient au moins un chiffre ;

— des constantes hexadécimales : `0x` suivi d'une suite non vide de chiffres entre `0` et `9`, de lettres entre `a` et`f` (minuscule ou majuscule) et d'underscores qui contient au moins un chiffre ou une lettre.

Ces constantes pourront être immédiatement suivies d'un suffixe qui sera le nom d'un type de base entier, par exemple `42i16`, ce qui aura pour effet de forcer leur type.

Les constantes flottantes seront :

— une constante décimale immédiatement suivie d'un suffixe `f32` ou `f64` ;

— une constante décimale immédiatement suivie d'un point ;

— une constante décimale immédiatement suivie d'un point et d'une constante décimale ;

— une constante décimale immédiatement suivie d'un exposant ;

— une constante décimale immédiatement suivie d'un point, d'une constante décimale et d'un exposant.

Un exposant est la lettre `e` ou `E` suivi d'un signe `+` ou `-` optionnel, suivi d'une suite non vide de chiffres entre `0` et `9` et d'underscores, qui contient au moins un chiffre. Dans les trois derniers cas, la constante peut être immédiatement suivie par `f32` ou `f64` pour forcer le type.

Une constante booléenne est `true` ou `false`.

Une expression est soit une expression sans bloc, soit une expression avec bloc.

Une expression sans bloc peut être :

— soit une constante ;

— soit un identifiant qui représentera une variable ;

— soit une expression précédée d'un symbole unaire parmi `&` (référence), `&mut` (référence mutable, `&` et `mut` peuvent être séparés par des espaces et des commentaires), `-` (moins unaire, seulement sur les entiers signés et les nombres à virgule flottante), `!` (négation logique sur les booléens et bit-à-bit sur les entiers) et `*` (déréférencement) ;

— soit deux expressions séparées par un symbole binaire parmi `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^` (xor), `<<` (décalage à gauche sur les entiers), `>>` (décalage à droite sur les entiers), `=`, `!=`, `<`, `<=`, `>`, `>=`, `&&`, `||` ;

— soit une conversion explicite $e$ `as` $t$ où $e$ est une expression et $t$ un type ;

— soit une affectation $x$ `=` $e$ où $x$ est un certain nombre (potentiellement nul) de déréférencements d'un identifiant `*`$\cdots$`*`$y$ et $e$ une expression ;

— soit un appel de fonction $e$`(`$ps$`)` où $e$ est une expression et $ps$ est soit vide, soit une liste d'expression séparées par `,`, avec optionnellement une `,` à la fin ;

— soit `continue` $lo$ où $lo$ est soit vide, soit une étiquette ;

— soit `break` $lo$ $eo$ où $lo$ est soit vide, soit une étiquette, et $eo$ est soit vide, soit une expression ;

— soit `return` $eo$ où $eo$ est soit vide, soit une expression ;

— soit une expression entre parenthèses.

Une expression avec bloc peut être

— soit $lo$ $b$ où $lo$ est soit vide, soit une étiquette suivie de `:`, et $b$ est un bloc (cf. infra) ;

— soit $lo$ `loop` $b$ où $lo$ est soit vide, soit une étiquette suivie de `:`, et $b$ est un bloc ;

— soit $lo$ `while` $e$ $b$ où $lo$ est soit vide, soit une étiquette suivie de `:`, $e$ est une expression et $b$ est un bloc ;

— soit `if` $e$ $b$ où $e$ est une expression et $b$ est un bloc ;

— soit `if` $e$ $b_1$ `else` $b_2$ où $e$ est une expression et $b_1$ est un bloc et $b_2$ est soit un bloc, soit une expression commençant par `if`.

Un bloc est constitué de `{` $s$ `}`, où $s$ est :

— soit vide ;

— soit une expression sans bloc ;

— soit une séquence de déclarations ;

— soit une séquence de déclarations suivie d'une expression sans bloc.

Une déclaration peut être :

— soit la déclaration vide `;` ;

— soit $e$ `;` où $e$ est une expression ;

— soit une déclaration de variable de la forme `let` $p$ $to$ $eo$`;` où $p$ est soit un identifiant, soit `mut` suivi d'un identifiant, $to$ est soit vide, soit un type et $eo$ est soit vide, soit `=` suivi d'une expression ;

— soit une déclaration de fonction.

Une déclaration de fonction est de la forme `fn` $x$ `(` $ps$ `)` $to$ $b$ où :

— $x$ est un identifiant ;

— $ps$ est soit vide, soit une séquence de paramètres séparés par des virgules, avec optionnellement une virgule à la fin, où chaque motif est de la forme $p$ `:` $t$ où $p$ est soit un identifiant, soit `mut` suivi d'un identifiant, et $t$ est un type ;

— $to$ est soit vide, soit `->` suivi d'un type (représentant le type de retour) ;

— $b$ est un bloc.

La priorité des opérateurs est donné sur la page https://doc.rust-lang.org/beta/reference/expressions.html#expression-precedence. On pourra ajouter à cela le fait que `{` est plus prioritaire que `return` et `break` pour éviter des conflits avec par exemple `while return {}`.

Un fichier Rustine contient une séquence de déclarations de fonction.

## 3 Questions

1. Définir des types de données correspondant à la syntaxe abstraite des programmes Rustine (couvrant toute la grammaire). En particulier on définira entre autres des types `type_`, `expression`, `declaration` et `function_`.
2. À l'aide de lex/yacc, ou ocamllex/ocamlyacc, écrire un analyseur lexical et syntaxique qui lit un fichier Rustine et qui retourne l'arbre de syntaxe abstraite associé (qui retourne donc une valeur de type `function_ list`).

3. Écrire une fonction `print_consts` qui prend en argument un AST de fichier Rustine et qui affiche toutes les constantes apparaissant dans le programme, en revenant à la ligne entre chaque. On affichera ces constantes de façon usuelle (`%g` de `printf` pour les nombre en virgule flottante par exemple). Ainsi `1_.2_34E8_1f32` sera affiché `1.234e+81`.

4. Écrire une fonction `check_scope` qui prend en argument un AST de fichier Rustine et qui vérifie que les identifiants ont bien été déclarés quand ils sont utilisés (variables et appels de fonctions).

   Quand une fonction est déclarée dans un bloc, elle est utilisable dans tout le bloc, y compris avant sa déclaration, et y compris dans son corps, ce qui permet des définitions (mutuellement) récursives.

   On ne peut pas déclarer deux fonctions avec le même nom dans un bloc. Par contre, il est possible de déclarer une fonction avec le même nom dans un sous-bloc.

   Les noms de variables peuvent être masqués : comme en OCaml, si on définit deux fois la même variable, la deuxième cache la première. On peut utiliser le même identifiant pour un nom de fonction et un nom de variable, dans ce cas la variable masque le nom de fonction.

   Attention, en Rust et donc en Rustine, dans le cas d'une fonction `g` imbriquée dans une fonction `f`, la fonction `g` n'a pas accès aux variables locales de `f` !

5. Écrire une fonction `check_affect` qui prend en argument un AST de fichier Rustine et qui vérifie que les affectations se font sur des variables ou paramètres déclarés mutables avec `mut`, ou sur un déréférencement $*v$ où $v$ a un type de la forme `&mut` $t$.

6. Écrire une fonction `check_type` qui prend en argument un AST de fichier Rustine et qui vérifie que les expressions sont bien typées. Il n'y a pas en Rustine de conversion implicite de types, si ce n'est qu'un `&mut` $t$ peut être utilisé comme un `&` $t$. Une constante entière aura par défaut le type `i32`, sauf si le contexte permet de préciser son type. (Par exemple, `let x : u8 = 25` ou bien `42 + 21i64`.)

   Pour déréférencer une expression $e$ ($*e$), il faut que $e$ ait un type `&`$t$ ou `&mut` $t$, et dans ce cas $*e$ a le type $t$. Si $e$ a le type $t$, `&`$e$ (resp. `&mut` $e$) a le type `&`$t$ (resp. `&mut` $t$).

   Dans une déclaration $e$`;`, $e$ peut avoir n'importe quel type, qui est ignoré.

   Les expressions $e$ dans `if` $e$ `{` ...`}` ou `while` $e$ `{` ...`}` doivent être de type `bool`. Le type d'un bloc est celui de l'expression sans bloc qui le termine si elle est présente, `()` sinon. Les blocs des `loop` et `while` doivent être de type `()`. Les blocs $b_1$ et $b_2$ d'un `if` $e$ $b_1$ `else` $b_2$ doivent être du même type, ce qui donne le type du `if`. Le type du bloc $b$ d'un `if` $e$ $b$ sans `else` doit être `()`, et le `if` aura ce type.

   Si le type de retour d'une fonction n'est pas précisé, c'est `()`. L'expression $e$ d'un `return` $e$`;` doit être du type de retour de la fonction. Un `return` sans expression n'est possible que si le type de retour est `()`.

   Toutes les expressions $e$ d'un `break` $e$`;` dans le bloc d'un `loop` doivent être du même type, ce qui donne le type du `loop`. Les `break` du bloc d'un while ne peuvent pas avoir d'expression, même si elle est de type `()`. C'est la même chose pour les `break`

qui ont une étiquette, en considérant le bloc, le `loop` ou le `while` référencé par l'étiquette.

Le `-` unaire ne peut s'appliquer que sur des entiers signés ou des nombres à virgule flottante. Le `!` peut s'appliquer aux entiers (non bit-à-bit) et à `bool`. Les opérateurs binaires doivent avoir le même type à gauche et à droite. Les opérateurs binaires `+ - * / %` ne peuvent s'appliquer que sur des entiers ou des flottants, et donnent le même type. Les opérateurs binaires `& | ^` ne peuvent s'appliquer que sur des entiers ou des `bool`, et donnent le même type. Les opérateurs binaires `<< >>` ne peuvent s'appliquer que sur des entiers et donnent le même type. Les opérateurs binaires `= != < <= > >=` peuvent s'appliquer à n'importe quel type, et donnent `bool`. Les opérateurs binaires `&& ||` ne s'appliquent qu'à des `bool` et donnent `bool`.

L'expression $e$ `as` $t$ aura le type $t$. Ceci n'est possible que si le type de $e$ et $t$ sont tous les deux numériques (par exemple `4u16 as f32`), ou si le type de $e$ est `bool` et $t$ un type entier.

7. Écrire un programme qui parse un fichier Rustine, affiche ses constantes à l'aide de la fonction `print_consts` puis effectue les vérifications `check_affect`, `check_scope`, et `check_type` en renvoyant un code d'erreur différent de 0 au cas où ils ne passent pas. Afficher des messages expliquant les erreurs sera fortement apprécié.