



ECOLE NATIONALE SUPERIEURE
D'INFORMATIQUE ET ANALYSE DES SYSTEMES

Réalisation d'un mini compilateur de langage de programmation Rust -Rapport-

Filière : Génie logiciel

Réalisé par :
Oussama KRAB
Idriss ZARKAOUI
Mohamed SALBI
Yahya BENZHA

Encadré par :
Pr. TABII Youness

[Année universitaire 2021 – 2022]

Remerciements

Nous désirons tout d'abord adresser toute notre gratitude à notre Professeur TABII Youness de nous avoir assuré le cours de compilation et les séances de travaux pratiques qui nous ont aidé à bien réaliser ce projet.

Table des matières

1	Présentation du langage Rust :	5
1.1	Inroduction	5
1.2	Grammaire Mini-Rust	5
1.2.1	Syntax	5
1.2.2	Conventions lexicales :	5
1.2.3	Règles de production	6
2	Réalisation et test de compilateur Minu-Rust	7
2.1	Chaîne de compilation	7
2.2	Réalisation	8
2.2.1	Structure du projet	8
2.2.2	Structures et Fonctions	8
2.2.3	Exemples de Test	10
2.2.4	Remarques générales	13
2.2.5	Obstacles et Défis	13
2.3	Conclusion	14

1 Présentation du langage Rust :

1.1 Introduction

Rust est un langage multiparadigme conçu et développé par Mozilla Research depuis 2010, il permet aux développeurs de créer des logiciels fiables et efficaces. Rust se concentre sur la sécurité et les performances, comme le C et le C++, et est rapide et efficace en termes de mémoire, sans garbage collection. Il s'intègre à d'autres langages et fonctionne également sur un système embarqué. La première version stable de Rust voit le jour en 2015.

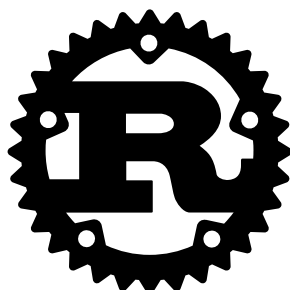


Figure 1.1: Logo de Rust

1.2 Grammaire Mini-Rust

1.2.1 Syntax

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)
$\langle \text{r\`egle} \rangle^+$	répétition de la règle $\langle \text{r\`egle} \rangle$ au moins une fois
$\langle \text{r\`egle} \rangle^?$	utilisation optionnelle de la règle $\langle \text{r\`egle} \rangle$ (i.e. 0 ou 1 fois)

1.2.2 Conventions lexicales :

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires peuvent prendre la forme : débutant par `//` et s'étendant jusqu'à la fin de la ligne.

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

$\langle \text{chiffre} \rangle ::= 0-9$

$\langle \text{alpha} \rangle ::= a-z \mid A-Z$

$\langle \text{ID} \rangle ::= \langle \text{alpha} \rangle (\langle \text{alpha} \rangle \mid \langle \text{chiffre} \rangle \mid _)^*$

Les identificateurs suivants sont des mots clés :

if else false true fn struct let mut const return struct while loop for break continue in

Les types considérés dans ce mini rust :

- $\langle \text{NUM_TYPE_TOKEN} \rangle$: il capture tous les types nombres : **u8, u16, u32, u64, u128** (unsigned) et **i8, i16, i32, i64, i128** (signed)
- $\langle \text{STRING_TYPE_TOKEN} \rangle$: **String**
- $\langle \text{BOOL_TOKEN} \rangle$: **bool**
- $\langle \text{UNIT_TOKEN} \rangle$: **()** : peut être associé à tout type

1.2.3 Règles de production

$\langle \text{DECL} \rangle$	$::= \langle \text{DECL_FON} \rangle \mid \langle \text{DECL_STRUCT} \rangle$
$\langle \text{DECL_STRUCT} \rangle$	$::= \text{struct } \langle \text{ID} \rangle \{ (\langle \text{ID} \rangle : \langle \text{TYPE} \rangle)^*, \}$
$\langle \text{DECL_FON} \rangle$	$::= \text{fn } \langle \text{ID} \rangle (\langle \text{ARG} \rangle^*,) (-> \langle \text{TYPE} \rangle)? \langle \text{BLOC} \rangle$
$\langle \text{TYPE} \rangle$	$::= \text{String} \mid \text{bool} \mid \langle \text{NUM_TYPE_TOKEN} \rangle \mid ()$
$\langle \text{ARG} \rangle$	$::= \langle \text{ID} \rangle : \langle \text{TYPE} \rangle$
$\langle \text{BLOC} \rangle$	$::= \{ \langle \text{INSTR} \rangle^* \langle \text{EXPR} \rangle? \}$
$\langle \text{INSTR} \rangle$	$::= \langle \text{EXPR} \rangle;$ $\mid \text{let mut? } \langle \text{ID} \rangle = \langle \text{EXPR} \rangle ;$ $\mid \text{let mut? } \langle \text{ID} \rangle = \langle \text{ID} \rangle \{ (\langle \text{ID} \rangle : \langle \text{EXPR} \rangle)^*, \};$ $\mid \text{while } \langle \text{EXPR} \rangle \langle \text{BLOC} \rangle$ $\mid \text{loop } \langle \text{BLOC} \rangle$ $\mid \text{for } \langle \text{EXPR} \rangle \langle \text{BLOC} \rangle$ $\mid \text{return } \langle \text{EXPR} \rangle? ;$ $\mid \langle \text{IF} \rangle$
$\langle \text{IF} \rangle$	$::= \text{if } \langle \text{EXPR} \rangle \langle \text{BLOC} \rangle (\text{else } (\langle \text{BLOC} \rangle \mid \langle \text{if} \rangle))?$
$\langle \text{EXPR} \rangle$	$::= \langle \text{NUM_TOKEN} \rangle \mid \text{true} \mid \text{false}$ $\mid \langle \text{ID} \rangle$ $\mid \langle \text{EXPR} \rangle \langle \text{OPR} \rangle \langle \text{EXPR} \rangle \mid \langle \text{NEG} \rangle \langle \text{EXPR} \rangle$ $\mid \langle \text{ID} \rangle (\langle \text{EXPR} \rangle^*,)$ $\mid \langle \text{ID} \rangle \text{ in } (\langle \text{NUM_TOKEN} \rangle .. \langle \text{NUM_TOKEN} \rangle)$ $\mid \text{println! } (\langle \text{ID} \rangle \mid \langle \text{CHAINE} \rangle)$ $\mid \langle \text{BLOC} \rangle$ $\mid \langle \text{IF} \rangle$ $\mid (\langle \text{EXPR} \rangle)$
$\langle \text{OPR} \rangle$	$::= == \mid != \mid < \mid <= \mid > \mid >= \mid + \mid - \mid * \mid / \mid \% \mid \&\& \mid \mid =$
$\langle \text{NEG} \rangle$	$::= - \mid !$
$\langle \text{CHAINE} \rangle$	$::= \text{''} \langle \text{ID} \rangle^* \text{''}$

2 Réalisation et test de compilateur Minu-Rust

2.1 Chaîne de compilation

Nous avons suivi tout au long de la réalisation de ce compilateur les analyseurs vus aux TP dans l'ordre suivant:

- Analyse lexicale : vérification que les caractères ou suites des caractères du programme sont des constituants d'un programme du langage cible (ici le langage C). Cette phase découpe le flux de caractères du fichier en une suite d'unités lexicales prise en compte par l'étape suivante.
- Analyse syntaxique : vérification que l'ordonnancement des unités lexicales respecte une configuration possible décrite par une grammaire.
- Analyse sémantique : vérification que les opérations décrites par la suite d'instructions ont un sens pour les données manipulées. Par exemple, comparer une donnée simple, comme un entier à un ensemble de données, comme les caractéristiques d'un individu, n'a pas de sens, même si la grammaire permet l'écriture d'une instruction semblable.

2.2 Réalisation

2.2.1 Structure du projet

Fichier	Utilité
main.c	Le fichier qui contient la fonction <i>main()</i> pour l'exécution du programme
lexique.h	Le fichier qui contient les déclarations des structures et les fonctions de l'analyse lexicale
lexique.c	Le fichier qui contient l'implémentation des fonctions déclarées en header pour la lecture des tokens
syntax.h	Le fichier qui contient les déclarations des structures et les fonctions de l'analyse syntaxique
syntax.c	Le fichier qui contient l'implémentation des fonctions déclarées en header pour la vérification de l'ordonnancement des tokens
semantique.h	Le fichier qui contient les déclarations des structures et les fonctions de l'analyse sémantique
semantique.c	Le fichier qui contient l'implémentation des fonctions déclarées en header pour l'étude formelle de la signification des programmes
erreurs.h	Le fichier qui contient les déclarations des structures et les fonctions de la gestion des erreurs
erreurs.c	Le fichier qui contient l'implémentation des fonctions de fichier header notamment de l'affichage
FILE.rs	Le fichier qui subit les tests durant la production

2.2.2 Structures et Fonctions

Dans toutes les déclarations on suit exactement la même organization et nomination vues aux TPs.

lexique.h

```
1 typedef struct {
2     char c;
3 } Symbol;
4
5 // L'énumération des classes lexicales
6 typedef enum {
7     FN_TOKEN, RETURN_TOKEN, RETURN_TYPE_TOKEN, CONST_TOKEN, LET_TOKEN,
```



```

8     MUT_TOKEN, IF_TOKEN, ELSE_TOKEN, WHILE_TOKEN, LOOP_TOKEN, FOR_TOKEN,
9     CONTI_TOKEN, BREAK_TOKEN, RANG_TOKEN, FALSE_TOKEN, TRUE_TOKEN, NON_TOKEN,
10    PV_TOKEN, PT_TOKEN, COL_TOKEN, PLUS_TOKEN, MOINS_TOKEN, STRUCT_TOKEN,
11    MULT_TOKEN, DIV_TOKEN, REST_TOKEN, OR_TOKEN, AND_TOKEN, VIR_TOKEN,
12    AFF_TOKEN, INF_TOKEN, INFEG_TOKEN, SUP_TOKEN, SUPEG_TOKEN, PRINT_TOKEN,
13    GUILL_TOKEN, DIFF_TOKEN, COMP_TOKEN, PO_TOKEN, PF_TOKEN, ACCOLO_TOKEN,
14    ACCOLF_TOKEN, CROCHO_TOKEN, IN_TOKEN, CROCHF_TOKEN, FIN_TOKEN,
15    BOOL_TOKEN, ERREUR_TOKEN, NUM_TOKEN, STRING_TYPE_TOKEN, NUM_TYPE_TOKEN,
16    UNIT_TOKEN, ID_TOKEN, END_TOKEN,
17 } CODES_LEX;
18
19 // type symbole courant
20 typedef struct {
21     CODES_LEX CODE;
22     char NOM[20];
23     int val;
24 } TSym_Cour;
25
26 // Les prototypes des fonctions :
27 void symbol_suivant();
28 void revenir_arriere();
29 char* to_string();
30 void assign_nom_token(char* mot);
31 void lire_mot();
32 void sauter_espace();
33 void lire_nombre();
34 void sauter_commentaire();
35 void lire_special();
36 void print_token();
37 void token_suivant();

```

syntax.h

```

1 void Test_Symbole(CODES_LEX cl, ERRORS_LEX er);
2 void Test_Symbole_ID(CODES_LEX cl, TSYM TIDF, ERRORS_LEX er);
3
4
5 void DECL(); // DECLARATION GLOBALE
6 void DECL_STRUCT(); // DECLARATION D'UNE STRUCTURE
7 void DECL_FON(); // DECLARATION D'UNE FONCTION
8 void TYPE(); // LE TYPE (POUR CE PETIT MINI RUST ON
9 //A PRIS EN COMPTE QUE DE 3 TYPES : bool,
10 //NUM_TYPE_TOKEN(DEFINIE EN LEXIQUE) et ():UNIT_TOKEN )
11 void ARG(); // ARGUMENTS DES FONCTIONS
12 void BLOC(int fun_indicator, int loops_indicator); // DEFINITION D'UN BLOC
13 void INSTR(int loops_indicator); // DEFINITION D'UNE INSTRUCTION ,
14 //ELLE CONTIENT AUSSI LES INST DY TYPE WHILE LOOP FOR ...
15 void IF(int indic); // BLOC IF ELSE ...
16 void WHILE(); // BLOC WHILE
17 void LOOP(); // BLOC LOOP
18 void FOR(); // BLOC FOR
19 void EXPR(); // DEFINITION D'UNE EXPRESSION
20 void ARG_VERI();
21 void OPR(); // DEFINITION D'UNE OPERATION
22 void PRINT(); // AFFICHAGE SUR CONSOLE : println!(...)
23 void NEG(); // DEFINITION DE LA NEGATION : ARITHMETIQUE (-) OU LOGIQUE (!)
24 int IS_OPR(); // FONCTION INDICATRICE SI UN TOKEN EST UN OPERATEUR

```

semantique.h

```
1 #define MAX_IDFS_NBRE 40
2 typedef enum {
3     TINSTR, // ce type est specialement pour distinguer
4     //les IDs lors des instructions
5     TMUT, // type MUTABLE : let mut a = 1;
6     TLET, // type LET : let a = 0;
7     TCONST, // type const let a : i32 = 6;
8     TFONC, // type FN : fn fonction() {...}
9     TSTRUCT, // type STRUCT : struct qwerty {...}
10    TARG, // type argument des fonctions
11 } TSYM;
12
13
14 typedef struct {
15     char NOM[20];
16     TSYM TIDF ;
17 } T_TAB_IDF;
18
19 // Prototypes des fonctions semantiques
20 int idf_existe();
21 void ajouter_idf(TSYM TIDF);
22 void mettre_a_jour_idf();
23 void mettre_a_jour_nbre();
```

erreurs.h

```
1 typedef enum {
2     FN_ERROR, RETURN_ERROR, RETURN_TYPE_ERROR, CONST_ERROR, LET_ERROR,
3     MUT_ERROR, IF_ERROR, ELSE_ERROR, WHILE_ERROR, LOOP_ERROR, FOR_ERROR,
4     CONT_ERROR, BREAK_ERROR, RANGE_ERROR, FALSE_ERROR, TRUE_ERROR, NON_ERROR,
5     PV_ERROR, PT_ERROR, COL_ERROR, PLUS_ERROR, MOINS_ERROR, STRUCT_ERROR,
6     MULT_ERROR, DIV_ERROR, REST_ERROR, OR_ERROR, AND_ERROR, VIR_ERROR,
7     AFF_ERROR, INF_ERROR, INFEG_ERROR, SUP_ERROR, SUPEG_ERROR, DIFF_ERROR,
8     COMP_ERROR, PO_ERROR, PF_ERROR, ACCOLO_ERROR, ACCOLF_ERROR,
9     CROCHO_ERROR, CROCHF_ERROR, FIN_ERROR, BOOL_ERROR, ERREUR_ERROR,
10    NUM_ERROR, NUM_TYPE_ERROR, STRING_TYPE_ERROR, UNIT_ERROR,
11    GUILL_ERROR, ID_ERROR, END_ERROR,
12    // complex errors :
13    TYPE_ERROR, INSTR_ERROR, EXPR_ERROR , DECL_ERROR, PRINT_ERROR,
14    // semantic errors :
15    REDECL_ID_ERROR, UNDECL_ID_ERROR, UNCHANGED_CONST_ERREUR, ILLEGAL_ARG
16 } ERRORS_LEX;
17
18 // fonction d'affichage des erreurs
19 void Erreur(ERRORS_LEX er, int counter);
```

2.2.3 Exemples de Test

On introduit le fichier de test final : FILE.rs

```
1 //voici des commentaires multiples
```

```

2 // deuxieme lignes
3 // troisieme ligne
4 struct Person {
5     nom: String,
6     age: u32,
7 }
8 fn main() {
9     let mut count = 0;
10    //let mut a = 189 ;
11    let mut alpha = ();
12    let mut n =0;
13    let mut a;
14    // Infinite loop
15    loop {
16        count = count + 1;
17
18        if count == 3 {
19            continue;
20        }
21        if count == 5 {
22            a = a -2;
23            break;
24        }
25    }
26    // A counter variable
27    let b = 1;
28    // Loop while 'n' is less than 101
29    while n != 101 {
30        if n % 15 == 0 {
31        } else if n % 3 == 0 {
32        } else if n % 5 == 0 {
33        } else {
34        }
35        // Increment counter
36        n = n + 1;
37    }
38    fn is_divisible_by(){
39
40    }

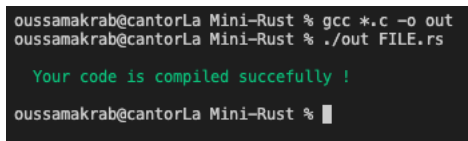
```

```

41 fn fizzbuzz(m : u32) -> bool {
42     if is_divisible_by(n,15) {
43         println!("abc");
44         m=m-1;
45     } else if is_divisible_by(n, 3) {
46         //un autre commentaire
47
48     } else if is_divisible_by(n, 5) {
49
50     } else {
51
52     }
53     //ceci est un commentaire
54 }
55 }
56 //Ceci est un commentaire final

```

Résultats



```

oussamakrab@cantorLa Mini-Rust % gcc *.c -o out
oussamakrab@cantorLa Mini-Rust % ./out FILE.rs

Your code is compiled succefully !

oussamakrab@cantorLa Mini-Rust %

```

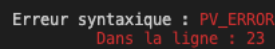
Figure 2.1: Résultat immédiat de l'exécution de fichier FILE.rs

On tente de modifier le code dans la ligne 23 en éliminant le point-virgule:

```

1 a = a -2

```



```

Erreur syntaxique : PV_ERROR
Dans la ligne : 23

```

Figure 2.2: Erreur syntaxique

De la même manière on fait exprès une redéclaration de la variable mutable **a** à la place de **b** dans la ligne 27 comme suit:

```

1 let a = 1;

```

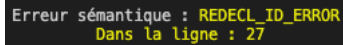


Figure 2.3: Erreur sémantique

2.2.4 Remarques générales

Tout au long de la production de ce mini compilateur de **Rust**, plusieurs choses sont à signaler :

❗ Dans ce mini Rust l'utilisation des structures dans les fonctions n'est pas valide

❗ La grammaire indiquée précédemment ne fait pas partie totalement dans un sens large à ce qui était produit et implémenté, la grammaire étant plein d'ambiguïtés pour cela nous avons utilisé des indicateurs (*loops_indicator*) et des blocs ($\langle ARG_VERI \rangle$) intermédiaires .

❗ Les types de nombres sont tous réduits à un seul token $\langle NUM_TYPE_TOKEN \rangle$

❗ Il est déconseillé de tester en général des programmes compilés correctement avec le compilateur Rust : ***rustc*** avec le compilateur Mini-Rust sauf si on sache quoi tester en tenant compte de la grammaire ci-dessus.

2.2.5 Obstacles et Défis

Un des plus grands obstacles s'était de comment choisir la grammaire afin de définir un petit compilateur qui puisse gérer et compiler des petits programmes *.rs*, C'est ainsi que la difficulté syntaxique et sémantique de **Rust** se révèle. La définition de certains types était d'ores et déjà exotique et nouvelle notamment :

```
1 let a = if(condition) {  
2 } else {  
3 }
```

On peut affecter un bloc tout entier à un identificateur !

L'un des défis aussi de la complétion de ce Mini-Rust est de concevoir

comment déclarer des structures définies et les utiliser au sein des fonctions.

2.3 Conclusion

Ce projet nous a permis d'aller plus loin pour savoir comment un compilateur fonctionne en réalité dans ses différentes phases d'analyse : lexicale, syntaxique et sémantique. Il nous a permis également de voir un exemple concret et d'appliquer les principes théoriques vus en cours. Nous avons désormais une vision globale plus claire sur les différentes étapes allant de l'écriture à l'exécution d'un programme.