Projet de programmation fonctionnelle et de traduction des langages

Année 2022/2023

Le but du projet de programmation fonctionnelle et de traduction des langages est d'étendre le compilateur du langage RAT réalisé en TP de traduction des langages pour traiter de nouvelles constructions : les **pointeurs**, le **bloc else optionnel** dans la conditionnelle, la **conditionnelle sous la forme d'un opérateur ternaire**, les **boucles "loop" à la Rust**.

Le compilateur sera écrit en OCaml et devra respecter les principes de la programmation fonctionnelle étudiés lors des cours, TD et TP de programmation fonctionnelle.

Table des matières

Préambule

- Le projet est à réaliser en binôme (même binôme qu'en TP).
- L'échange de code entre les différents binômes est interdit.
- Les sources fournies doivent compiler sur les machines des salles de TP.
- Les sources et le rapport sont à déposer sous Moodle avant le **jeudi 12 Janvier 2023 - 23h**. Aucun report ne sera accepté : anticipez !
- Les sources seront déposées sous forme d'une unique archive `<rat_xxx_yyy>.tar` où `xxx` et `yyy` sont les noms du binôme. Cette archive devra créer un répertoire `rat_xxx_yyy` (pensez à renommer le répertoire nommé `sourceEtu` donné en TP) contenant tous vos fichiers.
- le rapport (`rapport.pdf`) doit être dans le même répertoire, à la racine. Il n'est pas nécessairement long, mais doit expliquer les évolutions apportées au compilateur (voir section sur les critères d'évaluation) et les jugements de typages liés aux nouvelles constructions du langage.

1. $MAIN \rightarrow PROG$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ BLOC$
4. $PROG \rightarrow\ id\ \ BLOC$
5. $BLOC \rightarrow\ \{\ IS\ \}$
6. $IS \rightarrow\ I\ \ IS$
7. $IS \rightarrow$
8. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow\ A\ =\ E\ ;$
10. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
11. $I \rightarrow\ print\ \ E;$
12. $I \rightarrow\ if\ \ E\ \ BLOC\ \ else\ \ BLOC$
13. $I \rightarrow\ if\ \ E\ \ BLOC$
14. $I \rightarrow\ while\ \ E\ \ BLOC$
15. $I \rightarrow return\ E;$
16. $I \rightarrow loop\ BLOC$
17. $I \rightarrow id\ :\ loop\ BLOC$
18. $I \rightarrow break;$
19. $I \rightarrow break\ id;$
20. $A \rightarrow id$
21. $A \rightarrow (*\ A)$
22. $DP \rightarrow$
23. $DP \rightarrow TYPE\ id\ DP$
24. $TYPE \rightarrow\ bool$
25. $TYPE \rightarrow\ int$
26. $TYPE \rightarrow\ rat$
27. $TYPE \rightarrow\ TYPE\ *$
28. $E \rightarrow call\ id\ (\ CP\ )$
29. $CP \rightarrow$
30. $CP \rightarrow E\ CP$
31. $E \rightarrow\ [\ E\ /\ E\ ]$
32. $E \rightarrow\ num\ \ E$
33. $E \rightarrow\ denom\ \ E$
34. ~~$E \rightarrow\ id$~~
35. $E \rightarrow\ true$
36. $E \rightarrow\ false$
37. $E \rightarrow\ entier$
38. $E \rightarrow\ (\ E\ +\ \ E\ )$
39. $E \rightarrow\ (\ E\ *\ \ E\ )$
40. $E \rightarrow\ (\ E\ =\ \ E\ )$
41. $E \rightarrow\ (\ E\ <\ \ E\ )$
42. $E \rightarrow\ (\ E\ ?\ E : E\ )$
43. $E \rightarrow A$
44. $E \rightarrow null$
45. $E \rightarrow (new\ TYPE)$
46. $E \rightarrow\ \&\ id$

FIGURE 1 – Grammaire du langage RAT étendu

1 Extension du langage RAT

Le compilateur demandé doit être capable de traiter le langage RAT étendu comme spécifié dans la figure 1.

Le nouveau langage permet de manipuler :

1. des pointeurs ;
2. le bloc else optionnel dans la conditionnelle
3. la conditionnelle sous la forme d'un opérateur ternaire ;
4. les boucles "loop" à la Rust ;

1.1 Les pointeurs

RAT étendu permet de manipuler les pointeurs à l'aide d'une notation proche de celle de C :
— $A \rightarrow (* \ A)$: déréférencement : accès en lecture ou écriture à la valeur pointée par A ;
— $TYPE \rightarrow \ TYPE *$: type des pointeurs sur un type TYPE ;
— $E \rightarrow null$: pointeur null ;
— $E \rightarrow (new \ TYPE)$: initialisation d'un pointeur de type TYPE ;
— $E \rightarrow \& \ id$: accès à l'adresse d'une variable.

Le traitement des pointeurs a été étudié lors du dernier TD, il s'agit ici de coder le comportement défini en TD.

La libération de la mémoire n'est pas demandée.

Exemple de programme valide

```
main{
  int * px = (new int);
  (* px) = 42;
  print (*px);
  int x = 3;
  px = &x;
  int y = (*px);
  print y;
}
```

Ce programme affiche 423.

1.2 Le bloc else optionnel dans la conditionnelle

Nous souhaitons ne pas être obligé de fournir un bloc else à la conditionnelle. Une règle de production est ajoutée à la grammaire :
— $I \rightarrow \ if \ \ E \ \ BLOC$

Exemple de programme valide

```
test{
  int i = 0 ;
  while (i <11){
    if (denom [i/2] = 1){print i;}
    i = (i+1);
  }
}
```

Si la normalisation des rationnels est appelée à la construction d'un rationnel, ce programme doit afficher les nombres pairs entre 0 et 10.

1.3 La conditionnelle sous la forme d'un opérateur ternaire

Nous souhaitons, comme en C, permettre d'écrire la conditionnelle sous la forme d'un opérateur ternaire :

```
condition ? valeur_si_vrai : valeur_si_faux ;
```

Une règle de production est ajoutée à la grammaire :
— $E \rightarrow$ (E ? E : E)

Exemple de programme valide

```
int min (int a int b){
  return ((a<b)?a : b);
}

test{
  rat a =[2/3];
  rat y = ((denom a = 3) ? [1/3] : [0/3]);
  print ((call min ((num y) 1)=1) ? true : false );
}
```

1.4 les boucles "loop" à la Rust

Nous souhaitons ajouter au langage RAT, vu en cours, TD et TP, les boucles "**loop**" à la Rust [1].

Le mot-clé **loop** demande d'exécuter un bloc de code à l'infini ou jusqu'à ce que le programme soit arrêté manuellement, par un ctrl-c par exemple.

Exemple

```
main{
  loop { print 0; }
}
```

L'exécution de ce programme, affiche encore et encore 0 jusqu'à ce que le programme soit arrêté manuellement.

Break Il faut donc fournir un autre moyen de sortir d'une boucle en utilisant du code. Le mot-clé **break** à l'intérieur de la boucle demande au programme d'arrêter la boucle.

Exemple

```
main{
  int x = 0;
  loop {
    print x;
    if (x=10) {break;} else {x = (x+1);}
  }
}
```

L'exécution de ce programme, affiche les entiers de 0 à 10.

1. source : `https://jimskapt.github.io/rust-book-fr/ch03-05-control-flow.html`

Continue L'instruction **continue** est utilisée pour ignorer le reste de l'itération en cours et en débuter une nouvelle.

```
main{
  int c = 0;
  loop {
    c = (c+1);
    if (c=3){
      print 666;
      continue;
      }
    print c;
    if (c=5){
      break;
    }
  }
}
```

Ce programme affiche 1266645.

Boucles imbriquées S'il y a des boucles imbriquées dans d'autres boucles, **break** s'applique uniquement à la boucle la plus interne. Il est possible d'associer une *étiquette de boucle* à une boucle qu'il sera ensuite possible d'utiliser en association avec **break** pour préciser que ce mot-clé s'applique sur la boucle correspondant à l'étiquette plutôt qu'à la boucle la plus proche possible.

Exemples

```
main{
  int h = 0;
  heure : loop {
  int m = 0;
  loop {
    if (m=60){break ;}
    if (h=24){break heure;}
    print h;
    print m;
    m = (m+1);
    }
  h = (h+1);
  }
}
```

Ce programme affiche les heures de minuit (00) à 23h59 (2359).

Non - masquage Rust autorise :

— deux boucles de même nom au même niveau.

```
main{
  int h = 0;
  heure : loop {
```

```
    if (h=24){ break heure;}
    print h;
    h = (h+1);
  }
  h = 0 ;
  heure : loop {
    if (h=24){ break heure;}
    print h;
    h = (h+1);
  }
}
```

Affiche deux fois les nombres de 0 à 23.

— une boucle à avoir le même nom qu'une variable ou qu'une fonction.

```
main{
  int h = 0;
  h : loop {
    if (h=24){ break h;}
    print h;
    h = (h+1);
  }
}
```

Affiche les nombres de 0 à 23.

Rust lève un warning quand :

— deux boucles de même nom sont imbriquées (le break ou continue se réfère alors à la boucle du bon nom la plus proche en terme d'imbrication).

```
main{
  int d = 0;
  int u = 0;
  l : loop {
    if (d=10){ break l;}
    u = 0 ;
    l : loop {
      if (u=10){ break l;}
      print d; print u;
      u = (u+1);
    }
    d=(d+1);
  }
}
```

Affiche les nombres entre 00 et 99 sur deux décimales.

Le but est de se rapprocher le plus possible du comportement de Rust. En cas de doute, vous pouvez tester le comportement des programmes Rust, sans l'installer sur vos machines grâce à l'outil `https://play.rust-lang.org/`. Le programme présent au lancement de cette page est celui de l'affichage des heures.

Syntaxe du langage RAT étendu Pour traiter ces nouvelles boucles, les règles de grammaire suivantes sont ajoutées aux règles existantes :

1. $I \rightarrow loop\ BLOC$
2. $I \rightarrow id\ :\ loop\ BLOC$
3. $I \rightarrow break;$
4. $I \rightarrow break\ id;$
5. $I \rightarrow continue;$
6. $I \rightarrow continue\ id;$

1.5 Combinaisons des différentes constructions

Bien sûr ces différentes constructions peuvent être utilisées conjointement.

Exemple de programme valide

```
(int *) valeur (bool p (int *) a (int *) b ){
  return (p ? a : b);
}

int permute (int* p1 int* p2)
{
int sauve = (*p1);
(*p1) = (*p2);
(*p2) = sauve;
return 0;
}

prog {
  int a = 0;
  int b = 1;
  int i = 0;
  bool pair = true;
  v1 : loop {
    (int *) res = call valeur (pair &a &b);
    print (* res);
    pair = (pair = false);
    i = (i+1);
    if (i=10){ break v1;}
  }

  print 888;

  (int *) c = (new int);
  (* c) = 0;
  (int *) d = (new int);
  (* d) = 1;
  int j = 0;
```

```
  loop {
    print(* c);
    int  inutile  = call permute (c d);
    j  = (j+1);
    if  (j=10){break;}
  }
}
```

Ce programme doit afficher `01010101018880101010101`.

2 Travail demandé

Vous devez compléter le compilateur écrit en TP pour qu'il traite le langage RAT étendu. Vous devrez donc :

- modifier l'analyseur lexical (`lexer.mll`) pour ajouter les expressions régulières associées aux nouveaux terminaux de la grammaire ;
- modifier l'analyseur syntaxique (`parser.mly`) pour ajouter / modifier les règles de productions, et modifier si nécessaire les actions de construction de l'AST ;
- compléter les passes de gestion des identifiants, de typage, de placement mémoire et de génération de code.

Attention, il est indispensable de bien respecter la grammaire de la figure 1 pour que les tests automatiques qui seront réalisés sur votre projet fonctionnent.

D'un point de vue contrôle d'erreur, seules les vérifications de bonne utilisation des identifiants et de typage sont demandés. Les autres vérifications (déréférencement du pointeur null...) ne sont pas demandées.

3 Conseils d'organisation du travail

Il est conseillé d'attendre la fin des TP pour commencer à coder le projet (le dernier TD porte sur les pointeurs), néanmoins le sujet est donné au début des TP pour que vous commenciez à réfléchir à la façon dont vous traiterez les extensions et que vous puissiez commencer à poser des questions aux enseignants lors des TD / TP.

Il est conseillé de finir la partie demandée en TP et que tous les tests unitaires fournis passent avant de commencer le projet, afin de partir sur des bases solides et saines.

Il est conseillé d'ajouter les fonctionnalités les unes après les autres en commençant par les pointeurs dont le traitement aura été présenté lors du dernier TD.

Il est conseillé, pour chaque nouvelle fonctionnalité de procéder par étape :

1. compléter la structure de l'arbre abstrait issu de l'analyse syntaxique ;
2. modifier l'analyseur lexical, l'analyseur syntaxique et la construction de l'arbre abstrait ;
3. tester avec le compilateur qui utilise des "passes NOP" ;
4. compléter la structure de l'arbre abstrait issu de la passe de gestion des identifiants ;
5. modifier la passe de gestion des identifiants ;
6. tester avec le compilateur qui ne réalise que la passe de gestion de identifiants ;

7. compléter la structure de l'arbre abstrait issu de la passe de typage ;
8. modifier la passe de typage ;
9. tester avec le compilateur qui réalise la passe de gestion de identifiants et celle de typage ;
10. compléter la structure de l'arbre abstrait issu de la passe de placement mémoire ;
11. modifier la passe de placement mémoire ;
12. tester avec le compilateur qui réalise la passe de gestion de identifiants, celle de typage et de placement mémoire ;
13. modifier la passe de génération de code ;
14. tester avec le compilateur complet et itam.

4 Critères d'évaluation

Une grille critériée sera utilisée pour évaluer votre projet. Elle décrit les critères évalués pour le style de programmation, le compilateur réalisé et le rapport. Pour chacun d'eux est précisé ce qui est inacceptable, ce qui est insuffisant, ce qui est attendu et ce qui est au-delà des attentes.

Programmation fonctionnelle (40%)					
		Inacceptable	Insuffisant	Attendu	Au-delà
Compilation		Ne compile pas	Compile avec des warnings	Compile sans warning	
Style de programmation fonctionnelle		Le code est dans un style impératif	Il y a des fonctions auxiliaires avec accumulateurs non nécessaires	Les effets de bords ne sont que sur les structures de données et il n'y a pas de fonctions auxiliaires avec accumulateur non nécessaire	
Représentation des données		Type non adapté à la représentation des données	Type partiellement adapté à la représentation des données	Type adapté à la représentation des données	Monade
Lisibilité	Code source documenté	Aucun commentaire n'est donné	Seuls des contrats succincts sont donnés	Des contrats complets sont donnés	Des contrats complets sont donnés et des commentaires explicatifs ajoutés dans les fonctions complexes
	Architecture claire	Mauvaise utilisation des modules / foncteurs et fonctions trop complexes	Mauvaise utilisation des modules / foncteurs ou fonctions trop complexes	Bonne utilisation des modules / foncteurs. Bon découpage en fonctions auxiliaires	Introduction, à bon escient, de nouveaux modules / foncteurs
	Utilisation des itérateurs	Aucun itérateur n'est utilisé	Seul List.map est utilisé	Une variété d'itérateurs est utilisé à bon escient dans la majorité des cas où c'est possible	Une variété d'itérateurs est utilisé à bon escient dans la totalité des cas où c'est possible
	Respects des bonnes pratiques de programmation	Code mal écrit rendant sa lecture et sa maintenabilité impossible (par exemple : failwith au lieu d'exceptions significatives, mauvaise manipulation des booléens, mauvais choix d'identifiants, mauvaise utilisation du filtrage...)	Code partiellement mal écrit rendant sa lecture et sa maintenabilité difficile	Code bien écrit facilitant sa lecture et sa maintenabilité	Code limpide
Fiabilité	Tests unitaires	Aucun test unitaire	Tests unitaires des fonctions hors *analyse_xxx*, ne couvrant pas tous les cas de bases et les cas généraux	Tests unitaires des fonctions hors *analyse_xxx*, couvrant les cas de bases et les cas généraux	Tests unitaires de toutes les fonctions, couvrant les cas de bases et les cas généraux
	Tests d'intégration	Aucun test d'intégration ou ne couvrant pas les quatre passes	Tests d'intégrations, des passes de gestion des identifiants, typage et génération de code, non complet	Tests d'intégration complets des passes de gestion des identifiants, typage et génération de code	Tests d'intégration complets des quatre passes

Traduction des langages (40%)				
	Inacceptable	Insuffisant	Attendu	Au-delà
Grammaire	Non conforme à la grammaire du sujet	Partiellement conforme à la grammaire du sujet	Conforme à la grammaire du sujet	Plus complète que la grammaire du sujet
Fonctionnalités traitées intégralement	Aucune	Quelques-unes	Toutes celles du sujet	Fonctionnalités non demandées dans le sujet traitées correctement
Pointeurs	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Bloc else optionnel	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Conditionnelle ternaire	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Loop à la Rust	Non traité	Partiellement traité ou erroné	Complètement traité et correct (y compris les utilisations des identifiants autorisées par Rust sans warning)	Capable de traiter les programmes pour lesquels Rust lève des warnings

Le nombre de points accordés par fonctionnalité dépendra de la difficulté de celle-ci. Les fonctionnalités "Bloc else optionnel" et "Conditionnelle ternaire" sont simples à intégrer au compilateur, alors que les deux autres sont plus complexes.

Rapport (20%)				
	Inacceptable	Insuffisant	Attendu	Au-delà
Forme	Beaucoup d'erreurs de syntaxe et d'orthographe et mise en page non soignée.	Beaucoup d'erreurs de syntaxe et d'orthographe ou mise en page non soignée	Peu d'erreurs de syntaxe ou d'orthographe et mise en page soignée	Pas d'erreur de syntaxe ou d'orthographe et mise en page soignée
Introduction	Non présente	Copier / coller du sujet	Bonne description du sujet et des points abordés dans la suite du rapport	
Types	Aucune justification sur l'évolution de la structure des AST	Justifications non pertinentes ou non complètes sur l'évolution de la structure des AST	Justifications pertinentes et complètes sur l'évolution de la structure des AST	Justifications pertinentes et complètes sur l'évolution de la structure des AST. Comparaison avec d'autres choix de conception.
Jugement de typage	Non donnés	Partiellement donnés ou erronés	Complètement donnés et corrects	
Pointeurs	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Bloc else optionnel	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Conditionnelle ternaire	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Loop à la Rust	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Conclusion	Non présente	Creuse	Bon recul sur les difficultés rencontrées	Bon recul sur les difficultés rencontrées et améliorations éventuelles