Projet de programmation fonctionnelle et de traduction des langages

Année 2021/2022

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**, l'**opérateur d'assignation d'addition**, les **enregistrements** et les **types nommés**.

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 13 Janvier - 23h**.
— 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 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. l'opérateur d'assignation d'addition (+=) ;
3. des types nommés ;
4. les enregistrements ;

Le choix a été fait d'avoir une grammaire lisible et simple, mais cela implique la présence d'un lourd parenthésage des expressions et des types pour éviter les conflits au moment de la génération de l'analyseur syntaxique.

De même, pour simplifier la grammaire, les identifiants de type (tid) commencent par une lettre majuscule, alors que les autres identifiants (id) commencent par une minuscule.

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);
  int x = 3;
  px = &x;
  int y = (*px);
  print y;
}
```

1.2 L'opérateur d'assignation d'addition

RAT étendu permet de manipuler l'opérateur d'assignation d'addition (+=). Cet opérateur doit être surchargé pour accepter les mêmes types que l'addition.

Une règle de production est ajoutée à la grammaire :

— $I \rightarrow \ A \ += \ E \ ;$

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

FIGURE 1 – Grammaire du langage RAT étendu

Exemple de programme valide

```
main{
  int i = 3;
  i += 4;
  print i;
  rat r = [4/5];
  r += [1/8];
  print r;
}
```

1.3 Les types nommés

RAT étendu permet de définir et manipuler des types nommés :

— ~~*PROG → FUN PROG*~~

— $PROG \rightarrow TD\ FUN\ PROG$: définition globale (au programme) d'un type nommé ;

— $TD \rightarrow$

— $TD \rightarrow typedef\ tid\ =\ TYPE\ ;\ TD$

— $I \rightarrow typedef\ tid\ =\ TYPE\ ;$: définition locale (à un bloc) d'un type nommé ;

— $TYPE \rightarrow tid$: utilisation d'un type nommé.

La compatibilité de deux types se fait par comparaison de leurs structures et non de leurs noms.

Rappel : pour simplifier la grammaire, les identifiants de type (*tid*) commencent par une lettre majuscule, alors que les autres identifiants (*id*) commencent par une minuscule.

Exemple de programme valide avec des définitions locales.

```
main{
  typedef Int2 = int;
  Int2 a = 3;
  typedef PInt = int *;
  PInt px = &a ;
  int y = (*px);
  print y;
}
```

Observer que `Int2` et int sont compatibles, et de même pour `Pint`, `int*` et Int2*.

Exemple de programme valide avec des définitions globales.

```
typedef Int2 = int;
Int2 plus (Int2 a Int2 b){
  return (a+b);
}
main{
  Int2 a = 3;
  Int2 b = 4;
  int y = call plus (a b);
  print y;
}
```

1.4 Les enregistrements

RAT étendu permet de définir et manipuler les enregistrements :

— $TYPE \rightarrow struct \{ DP \}$: définition d'un enregistrement (DP est une liste des types et noms des champs) ;

— $A \rightarrow (A.id)$: accès à un champ de l'enregistrement

— $E \rightarrow \{ CP \}$: création d'un enregistrement avec la liste des valeurs de ses champs

Exemple de programme valide sans combiner avec les types nommés :

```
main{
  struct {int x rat y} p = {3 [3/4]};
  print (p.x);
  (p.y) = [1/((p.x)+1)];
  print (p.y);
}
```

Le programme doit afficher `3[1/4]`.

Exemples de programmes valides en combinant avec les types nommés :

```
typedef Point2D = struct {int x int y} ;
typedef VecteurTranslation = struct {int dx int dy} ;
Point2D translate (Point2D p VecteurTranslation v){
  return {((p.x)+(v.dx)) ((p.y)+(v.dy))};
}
main{
  Point2D p1 = {3 4};
  Point2D p2 = {2 1};
  VecteurTranslation v = {1 4};
  Point2D p1t = call translate (p1 v);
  Point2D p2t = call translate (p2 v);
  print (p1t.x);
  print (p1t.y);
  print (p2t.x);
  print (p2t.y);
}
```

Le programme doit afficher `4835`.

```
typedef R1 = struct {int x int y};
typedef R2 = struct {int z R1 r};
int r2z (R2 r){ return (r.z); }
int r2x (R2 rec){ return ((rec.r).x); }
main{
  R2 a = {1 {2 3}};
  print (call r2z (a));
  print (call r2x (a));
}
```

Le programme doit afficher `12`.

Pour simplifier les analyses, deux structures différentes ne pourront pas avoir des champs de mêmes noms. Par exemple, le programme suivant devra être rejeté avec l'exception `DoubleDeclaration` :

```
main {
  struct {int x int y} p2D = {3 4};
  struct {int x int y bool z} p = {1 5 true};
  print (p2D.x);
}
```

Les règles usuelles de gestion des doubles déclarations et du masquage seront appliquées. Par exemple :

— le programme suivant sera rejeté pour double déclaration de x :

```
main {
  struct {rat x int y} p = {[1/3] 4};
  int x = 2;
}
```

— le programme suivant sera accepté :

```
main{
  struct {int x int y} p = {2 3};
  if ((p.x) < 4) {
    int x = 3;
    print x;
  } else {
    int x = 4;
    print x;
  }
}
```

— le programme suivant sera rejeté avec l'exception `MauvaiseUtilisationIdentifiant` de x (x comme variable masque x comme nom de champ) :

```
main{
  struct {int x int y} p = {2 3};
  if ((p.x) < 4) {
    int x = 3;
    print (p.x);
  } else {
    int x = 4;
    print (p.x);
  }
}
```

BONUS Un bonus sera accordé aux projets qui gèrent les types récursifs.

```
typedef ListeChainee = struct {int val ListeChainee* suivante};

int printListe (ListeChainee l){
  if ((l.suivante) = null){
  } else {
    print (l.val);
    ListeChainee s = (*(l.suivante));
    int inutil = call printListe (s) ;
  }
  return 0;
}

main {
  Listechainee v3 = {3 null};
  ListeChainee v2 = {2 (&v3)};
  ListeChainee v1 = {1 (&v2)};
  int inutil = call printListe (v1);
}
```

1.5 Combinaisons des différentes constructions

Bien sûr ces différentes constructions peuvent être utilisées conjointement.

Exemple de programme valide

```
typedef Droite = struct {rat pente rat ordonneOrigine} ;
typedef Position = struct {rat* x rat* y} ;
typedef PositionOption = struct {bool empty Position val} ;

rat divRat (rat n rat d){
  return [(num n * denom d)/(num d * denom n)];
}

rat moinsRat (rat r1 rat r2){
  return (r1 + ([-1/1] * r2));
}


bool eqRat (rat r1 rat r2){
  // artifice pour être sur qu'il n'y a pas de soucis avec la normalisation / + appelle la fonction de normalisation
  rat m = call moinsRat (r1 r2);
  return (num m = 0);
}

PositionOption intersection (Droite d1 Droite d2){
  if (call eqRat ((d1.pente) (d2.pente))){
    return {true {null null}};
```

```
  }else{
    rat* pxi = (new rat);
    rat xi = call divRat (
                    (call moinsRat ((d1.ordonneOrigine) (d2.ordonneOrigine)))
                    (call moinsRat ((d2.pente) (d1.pente))));
    (*pxi) = xi;
    rat* pyi = (new rat);
    (*pyi) = (((d1.pente)*xi)+(d1.ordonneOrigine));
    Position i = {pxi pyi};
    return {false i};
  }
}

bool printPositionOption (PositionOption po){
  if (po.empty){
    print false;
  } else {
    print (*((po.val).x));
    print (*((po.val).y));
  }
  return (po.empty);
}

main {
  Droite d1 = {[2/13] [1/3]};
  rat a = [1/13];
  rat b = [1/4];
  int i = 1 ;
  while (i < 4){
    Droite d2 = {a b};
    PositionOption po = call intersection (d1 d2);
    bool e = call printPositionOption (po);
    a += [1/13];
    b += [1/7];
    i += 1;
  }

}
```

Ce programme doit afficher `[13/-12][1/6]false[-221/84][1/-14]`.

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	
+=	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Types nommés	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Enregistrements	Non traité	Partiellement traité ou erroné	Complètement traité et correct	Gestion des enregistrements récursifs

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.
+=	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.
Types nommés	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.
Enregistrements	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