Projet de programmation fonctionnelle et de traduction des langages

Année 2023/2024

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**, les **tableaux**, les **boucles "for"** et les **goto**.

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

Preambule

— Le projet est à réaliser en binôme (même binôme qu'en TP).
— L'échange de code entre 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 18 Janvier - 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 Extension du langage RAT

Le compilateur demandé doit être capable de traiter le langage RAT étendu comme spécifié dans la figure 1. Pour simplifier la lecture, les ' ' sont omises autour des caractères, la répétition est noté $\star$ à distinguer du caractère '*' et le parenthésage EBNF est réalisé avec $\langle$ et $\rangle$.

Le nouveau langage permet de manipuler :

1. des pointeurs ;
2. des tableaux ;
3. des boucles "for" ;
4. des goto.

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.

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 = 423;
  px = &x;
  int y = (*px);
  print y;
}
```

Ce programme affiche 423.

1.2 Les tableaux

RAT étendu permet de manipuler les tableaux à l'aide d'une notation proche de celle de C :
— $A \rightarrow (A [E])$: accès en lecture ou écriture à la case d'indice E du tableau A ;
— $TYPE \rightarrow TYPE\ []$: type des tableaux de TYPE (la syntaxe est différente de celle de C pour simplifier l'écriture de la grammaire) ;
— $E \rightarrow (new\ TYPE [E])$: création d'un tableau de TYPE et de taille E.
— $E \rightarrow \{ CP \}$: initialisation d'un tableau avec un ensemble de valeurs.

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

FIGURE 1 – Grammaire (EBNF) du langage RAT étendu

Exemple de programme valide

```
main{
  rat [] t2 = {[1/7],[1/8],[1/9]};
  rat [] [] t = {(new rat [(1+2)]),{[1/4],[1/5],[1/6]}, t2};
  rat [] t0 = (t [0]);
  (t0 [0]) = [1/2];
  ((t [0])[1]) = [1/4];
  print (((t [0])[0]) + ((t [1])[0]));
  print (((t [0])[1]) + ((t [2])[2]));
}
```

Avec la normalisation des rationnels, il doit afficher [3/4][13/36].

1.3 Les boucles "for"

RAT étendu permet d'écrire des boucles "for" à l'aide d'une notation proche de celle de C :

— $I \rightarrow for\ (\ int\ id\ =\ E\ ;\ E\ ;\ id\ =\ E)\ BLOC$

Le premier paramètre correspond à la définition et l'initialisation de l'indice de boucle. Le second paramètre correspond à la condition d'arrêt de la boucle. Le dernier paramètre correspond à l'évolution de l'indice de boucle (souvent incrémenté ou décrémenté de 1 mais une expression quelconque est autorisée).

Exemple de programme valide

```
main{
  for (int x = 0 ; (x < 5) ; x = (x+1)) {
    print x;
  }
}
```

1.4 Goto

RAT étendu permet d'écrire des sauts à l'aide de l'instruction **goto**. Cette instruction permet de sauter à un point précis du programme déterminé à l'avance. Pour ce faire, des instructions peuvent être marquées à l'aide d'étiquettes. Une étiquette est un nom suivi du caractère ':'.

— $I \rightarrow goto\ id\ ;$: à l'exécution, l'instruction suivante sera celle marquée par l'identifiant ;

— $I \rightarrow id\ :$: marque l'instruction qui suit comme destination possible d'un **goto**.

Pour être utilisée avec un **goto**, une étiquette doit être déclarée c'est-à-dire qu'elle doit être utilisée pour marquer une instruction. La déclaration et l'utilisation doivent rester au sein du bloc principal ou d'une même fonction. Attention, une étiquette peut être utilisée avant d'être déclarée. En effet, le programme suivant est valide.

```
main {
 cst a = 4;
 if (a < 5) {
   goto etiq_then;
 } else {
   goto etiq_else ;
}
```

```
etiq_then :
  print true ;
  goto fin;

etiq_else :
  print false ;
  goto fin ;

fin :
}
```

Une étiquette peut avoir le même identifiant qu'une variable, constante ou fonction mais il ne peut pas y avoir deux étiquettes du même nom dans une même fonction ou dans le programme principal, même dans des blocs différents. Par exemple, le programme suivant devra être rejeté.

```
main {
 const a = 40;
 if (a < 5) {
   etiq :
   print a ;
 } else {
   etiq :
   print a ;
}
goto etiq ;
}
```

Ces sauts peuvent conduire à des programmes avec des comportements non spécifiés, que nous ne chercheront pas à rejeter (dans cet exemple, lors d'un saut direct à fin, la variable x n'a jamais été affectée).

```
main {
 const a = 40;
 if (a < 5) {
   goto suite;
 } else {
   goto fin;
}
suite :
  int x = (a + 4);

fin :
  print x;
}
```

Néanmoins, le programme suivant doit compiler et s'exécuter sans erreur et afficher 3.

```
main {
 const a = 40;
 if (a < 5) {
   goto suite;
 } else {
```

```
    goto fin;
}
suite :
  int x = (a + 4);

fin :
  int z = 3;
  print z;
}
```

Plus subtilement, le programme suivant doit compiler et s'exécuter sans erreur et afficher 76. Une attention particulière devra être portée dans le rapport à ce type de programme : explication de la problématique et de la solution mise en place.

```
main {
int i = 3 ;
goto etiq ;
while (i < 10) {
  int x = 5 ;
  print x ;
  etiq :
    int a = 7;
    print a;
    goto fin;
 }
fin :
  int j = 6;
  print j;
}
```

Une erreur pour double déclaration de x doit être levée lors de la compilation du programme suivant.

```
main {
 const a = 40;
 if (a < 5) {
   goto suite;
 } else {
   goto fin;
}
suite :
  int x = (a + 4);

fin :
  int x = 3;
  print x;
}
```

Remarque : Bien qu'utile dans certaines circonstances, l'instruction **goto** est fortement décriée, principalement pour deux raisons :

— mis à part dans des cas spécifiques, il est possible de réaliser la même action de manière plus claire à l'aide de structures de contrôles (Tant Que et For) ;
— l'utilisation de cette instruction peut amener votre code à être plus difficilement lisible et, dans les pires cas, en faire un code spaghetti.

Mais c'est un très bon cas d'étude pour tester votre compréhension d'un compilateur.

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 ){
  if p {
    goto a;
  } else {
    goto b;
  }
  a :
  return a;
  b :
  return b;
}

prog {
  const t = 5;
  (int *)[] tab = (new (int *)[t]);
  bool pair = true;
  int a = 0;
  int b = 1;
  for (int i = 0 ; (i < t) ; i = (i+1)){
    (tab[i]) = call valeur (pair &a &b);
    pair = (pair = false);
  }
  for (int i = 0 ; (i < t) ; i = (i+1)){
    print (*(tab[i]));
  }
}
```

Ce programme doit afficher `01010`.

2 Travail demandé

Vous devez compléter le compilateur écrit en TP pour qu'il traite le langage RAT étendu. Vous devrez donc 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 (indice de tableaux hors bornes, déréférencement du pointeur null, boucle infinie, ...) ne sont pas demandées.

3 Conseil 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 puisque leur traitement aura été vu 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 la grammaire et construire 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, pour le style de programmation, le compilateur réalisé et le rapport, les critères évalués. 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	
Tableaux	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Boucles "for"	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Goto	Non traité	Partiellement traité ou erroné	Complètement traité et correct	

Le nombre de points accordés par fonctionnalité dépendra de la difficulté de celle-ci. La fonctionnalité "Boucles "for"" est plus simple que les trois autres.

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.
Tableaux	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.
Boucles "for"	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.
Goto	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