

Projet de programmation fonctionnelle et de traduction des langages

Année 2025/2026

Le but du projet de programmation fonctionnelle et de traduction des langages est d'étendre le compilateur du langage **RAT**, réalisé lors des travaux pratiques de traduction des langages, afin de traiter de nouvelles constructions : les **pointeurs**, les **procédures**, le **passage de variables par référence** et les **types énumérés**.

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

1	Extension du langage RAT	2
1.1	Les pointeurs	3
1.2	Les procédures	3
1.3	Le passage des paramètres par références	4
1.4	Les types énumérés	6
1.5	Combinaison des différentes constructions	7
2	Travail demandé	7
3	Conseil d'organisation du travail	8
4	Critères d'évaluation	9

Preamble

- 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 sur **Moodle** avant le **jeudi 15 janvier - 23h**.
Aucun report ne sera accepté : anticipez !
- Les sources seront déposées sous la forme d'une unique archive `<rat_xxx_yyy>.tar` où `xxx` et `yyy` sont les noms des membres du binôme. Cette archive devra créer un répertoire `rat_xxx_yyy` (pensez à renommer le répertoire nommé `sourceEtu` fourni en TP) contenant tous vos fichiers.
- Le rapport (`rapport.pdf`) doit se trouver dans le même répertoire, à la racine. Il n'est pas nécessairement long, mais doit expliquer les évolutions apportées au compilateur (voir la section sur les critères d'évaluation) ainsi que les jugements de typage liés aux nouvelles constructions du langage.

1 Extension du langage RAT

Le compilateur demandé doit être capable de traiter le langage **RAT** étendu avec :

1. des **pointeurs** ;
2. des **procédures** ;
3. des **paramètres passés par référence** ;
4. des **types énumérés**.

La grammaire vue en TP est modifiée comme indiqué ci-dessous :

- | | |
|---|---------------------------------------|
| 1. $PROG' \rightarrow ENUM \star PROG \$$ | 5. $A \rightarrow id$ |
| 2. $ENUM \rightarrow enum\ tid\ \{IDS\} ;$ | 6. $ (\star A)$ |
| 3. $IDS \rightarrow tid\ \langle ,\ tid \rangle \star$ | 7. $E \rightarrow id\ (CP)$ |
| 4. $PROG \rightarrow FUN \star id\ BLOC$ | 8. $ [E / E]$ |
| 5. $FUN \rightarrow TYPE\ id\ (DP) BLOC$ | 9. $ num\ E$ |
| 6. $BLOC \rightarrow \{ I \star \}$ | 10. $ denom\ E$ |
| 7. $I \rightarrow TYPE\ id = E ;$ | id |
| $id = E ;$ | 11. $ A$ |
| 8. $ A = E ;$ | 12. $ true$ |
| 9. $ const\ id = entier ;$ | 13. $ false$ |
| 10. $ print\ E ;$ | 14. $ entier$ |
| 11. $ if\ E\ BLOC\ else\ BLOC$ | 15. $ (E + E)$ |
| 12. $ while\ E\ BLOC$ | 16. $ (E \star E)$ |
| 13. $ return\ E ;$ | 17. $ (E = E)$ |
| 14. $ return ;$ | 18. $ (E < E)$ |
| 15. $ id\ (CP) ;$ | 19. $ (E)$ |
| 16. $DP \rightarrow \Lambda$ | 20. $ null$ |
| 17. $ ref? TYPE\ id\ \langle ,\ ref? TYPE\ id \rangle \star$ | 21. $ (new\ TYPE)$ |
| 18. $TYPE \rightarrow bool$ | 22. $ \& id$ |
| 19. $ int$ | 23. $ ref\ id$ |
| 20. $ rat$ | 24. $ tid$ |
| 21. $ TYPE \star$ | 25. $CP \rightarrow \Lambda$ |
| 22. $ void$ | 26. $ E\ \langle ,\ E \rangle \star$ |
| 23. $ tid$ | |

Pour simplifier la lecture, les guillemets ' ' sont omis autour des caractères. Les symboles issus de l'EBNF sont notés en gris :

- la répétition est notée \star , à distinguer du caractère '*' ;
- l'optionnalité est notée $?$;
- le parenthésage est réalisé avec \langle et \rangle .

1.1 Les pointeurs

Le langage **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 en écriture à la valeur pointée ;
- $TYPE \rightarrow TYPE *$: 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 procédures

Nous souhaitons ajouter au langage **RAT** la possibilité d'écrire des procédures. Les procédures ne retournent pas de résultat. Elles ne produisent que des effets de bord, comme, par exemple, des affichages.

Les règles suivantes sont ajoutées à la grammaire de **RAT** :

- $TYPE \rightarrow void$: ajout du type **void** pour spécifier le type de retour d'une procédure ;
- $I \rightarrow id\ (CP)$; : appel à une procédure ;
- $I \rightarrow return$; : retour sans paramètre pour indiquer la fin d'une procédure.

Exemple de programme valide

```
void printRAdd(rat a, rat b){  
  print a;  
  print b;  
  print (a+b);  
  return ;  
}  
  
main{  
  rat x = [1/2];  
  rat y = [3/4];  
  printRAdd(x,y);  
}
```

Ce programme doit afficher [1/2] [3/4] [5/4].

1.3 Le passage des paramètres par références

Nous souhaitons ajouter au langage **RAT** la possibilité de passer les paramètres par référence. Dans le passage par valeur (ce qui a été fait en TP pour **RAT**), une copie des arguments réels est transmise aux paramètres formels correspondants. En revanche, dans le passage par référence, c'est l'adresse (l'emplacement en mémoire) des arguments réels qui est transmise aux paramètres formels. Toute modification effectuée sur ces derniers se répercutera donc directement sur les arguments réels.

Nous utiliserons une syntaxe à la C#. Les règles suivantes sont modifiées ou ajoutées à la grammaire de **RAT** :

- $DP \rightarrow \text{ref? } TYPE \text{ id } \langle, \text{ref? } TYPE \text{ id } \rangle \star$: ajout du mot-clé **ref** sur les paramètres passés par référence lors de la définition d'une fonction ;
- $E \rightarrow \text{ref } id$: indique les paramètres passés par référence lors de l'appel de fonction.

Exemple de programme valide

```
int p (ref int a, int b){
  a = (a+1);
  return (a+b);
}

test{
  int x = 0;
  int r = p (ref x,3);
  print r;
  print x;
}
```

Ce programme doit afficher 41.

Cas d'erreur

Une erreur doit être générée lorsqu'un paramètre déclaré comme passé par référence dans la définition d'une fonction ne l'est pas lors de son appel, et inversement. Par exemple, les programmes suivants doivent générer une erreur :

```
int p (ref int a, int b){
  a = (a+1);
  return (a+b);
}

test{
  int x = 0;
  int r = p (x,3);
  print r;
  print x;
}
```

```
int p (int a, int b){
  a = (a+1);
  return (a+b);
}

test{
  int x = 0;
  int r = p (ref x,3);
  print r;
  print x;
}
```

Enchaînement de passages par référence

Il est possible d'enchaîner les passages par référence, notamment en effectuant des appels de fonctions à l'intérieur d'autres fonctions, ou en définissant des fonctions récursives dont certains paramètres sont transmis par référence. Par exemple, le programme suivant est correct :

```
int p0(ref int a){
    a = (a+1);
    return 0;
}

int p0b(int a){
    a = (a+1);
    return 0;
}

int p1(ref int a){
    return (p0 (ref a));
}

int p2(ref int a){
    return (p1 (ref a));
}

test{
    int x = 55;
    print x;

    int r = p0(ref x);
    print x;

    r = p0b(x);
    print x;

    r = p1(ref x);
    print x;

    r = p2(ref x);
    print x;
}
```

Il doit afficher 5556565758.

1.4 Les types énumérés

RAT étendu permet de définir des types énumérés à l'aide d'une notation proche de celle de **C** :

- $MAIN \rightarrow ENUM \star PROG$: ajout de la possibilité de définir des types énumérés en début d'un programme **RAT** ;
- $ENUM \rightarrow enum\ tid\ \{IDS\}$; : définition d'un type énuméré avec son identifiant et la liste de ses valeurs ;
- $IDS \rightarrow tid\ \langle ,\ tid \rangle \star$: définition des différentes valeurs d'un type énuméré (séparées par des " , ");
- $TYPE \rightarrow tid$: utilisation d'un type énuméré ;
- $E \rightarrow tid$: valeur d'un type énuméré.

Les identifiants (*tid*) de type énuméré ainsi que leurs valeurs commencent par une lettre majuscule, alors que les autres identifiants (*id*) commencent par une minuscule.

Deux types énumérés ne peuvent pas avoir le même nom et une même valeur ne peut pas appartenir à deux types énumérés différents.

Il faudra surcharger l'opérateur "=" pour tester l'égalité de deux expressions de type énuméré.

Exemple de programme valide

```
enum Mois {Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet, Aout, Septembre, Octobre, Novembre, Decembre};
enum Jour {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche};

bool and (bool b1, bool b2){
    bool r = b1;
    if (r){
        return b2;
    }else {
        return false;
    }
}

bool estDateRenduProjet (Jour j, int d, Mois m, int a){
    bool b1 = (j==Jeudi);
    bool b2 = (d==15);
    bool b3 = (m==Janvier);
    bool b4 = (a==2026);
    return and(and(b1,b2),b3),b4);
}

test {
    print (estDateRenduProjet (Mercredi, 5, Novembre, 2025) );
    print (estDateRenduProjet (Jeudi, 15, Janvier, 2026) );
}
```

Ce programme doit afficher `false``true`.

1.5 Combinaison des différentes constructions

Bien sûr, ces différentes constructions peuvent être utilisées conjointement.

Exemple de programme valide

```
enum Operation
{Addition, Multiplication, Echange};

void echanger (ref int x, ref int y){
    int tmp = x;
    x=y;
    y=tmp;
    return;
}

void appliquerOperation
(Operation op, ref int x, ref int y){
    if (op=Addition){
        x = (x+y);
    } else {
        if (op=Multiplication){
            x = (x*y);
        } else {
            if (op=Echange){
                echanger(ref x, ref y);
            } else{}
        }
    }
    return;
}

void afficherValeurPointee (int* p){
    print (*p);
    return;
}

int* initialiserPointeur (int v){
    int* p = (new int);
    *p = v;
    return p;
}

main{
    int a = 10;
    int b = 5;
    int* p1 = &a;
    int* p2 = &b;
    int* p3 = initialiserPointeur (10);
    afficherValeurPointee (p1);
    afficherValeurPointee (p2);
    afficherValeurPointee (p3);

    appliquerOperation(Addition, ref a, ref b);
    afficherValeurPointee (p1);
    afficherValeurPointee (p2);
    afficherValeurPointee (p3);

    appliquerOperation(Echange, ref a, ref b);
    afficherValeurPointee (p1);
    afficherValeurPointee (p2);
    afficherValeurPointee (p3);

    appliquerOperation(Multiplication, ref a, ref b);
    afficherValeurPointee (p1);
    afficherValeurPointee (p2);
    afficherValeurPointee (p3);
}
```

Ce programme doit afficher 105101551051510751510.

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 fournie pour que les tests automatiques 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 demandées. Les autres vérifications, par exemple le déréférencement d'un pointeur **null**, 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 afin que vous puissiez réfléchir à la façon dont vous traiterez les extensions et poser des questions aux enseignants lors des TD/TP.

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

Il est également conseillé d'ajouter les fonctionnalités les unes après les autres.

1. Introduction des affectables et **uniquement** des affectables :
 - (a) Remplacer les règles $I \rightarrow id = E$; et $E \rightarrow id$ par les règles $A \rightarrow id$, $I \rightarrow A = E$; et $E \rightarrow A$.
 - (b) Modifier les AST et toutes les passes (introduction des méthodes d'analyse des affectables).
 - (c) Vérifier que les tests des TP passent.
2. Pour chaque nouvelle fonctionnalité, procéder par étapes :
 - (a) Compléter la structure de l'arbre abstrait issu de l'analyse syntaxique ;
 - (b) Modifier la grammaire et construire l'arbre abstrait ;
 - (c) Tester avec le compilateur qui utilise des "passes NOP" ;
 - (d) Écrire les tests de la passe de gestion des identifiants ;
 - (e) Compléter la structure de l'arbre abstrait issu de la passe de gestion des identifiants ;
 - (f) Modifier la passe de gestion des identifiants ;
 - (g) Tester avec le compilateur qui ne réalise que la passe de gestion des identifiants ;
 - (h) Écrire les tests de la passe de typage ;
 - (i) Compléter la structure de l'arbre abstrait issu de la passe de typage ;
 - (j) Modifier la passe de typage ;
 - (k) Tester avec le compilateur qui réalise la passe de gestion des identifiants et celle de typage ;
 - (l) Écrire les tests de la passe de placement mémoire ;
 - (m) Compléter la structure de l'arbre abstrait issu de la passe de placement mémoire ;
 - (n) Modifier la passe de placement mémoire ;
 - (o) Tester avec le compilateur qui réalise les passes de gestion des identifiants, de typage et de placement mémoire ;
 - (p) Écrire les tests de la passe de génération de code ;
 - (q) Modifier la passe de génération de code ;
 - (r) Tester avec le compilateur complet, et ainsi de suite.

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, il est précisé ce qui est inacceptable, ce qui est insuffisant, ce qui est attendu et ce qui dépasse les 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 <i>analyse_xxx</i> , ne couvrant pas tous les cas de bases et les cas généraux	Tests unitaires des fonctions hors <i>analyse_xxx</i> , 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	
Procédures	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Passage par référence	Non traité	Partiellement traité ou erroné	Complètement traité et correct	
Types énumérés	Non traité	Partiellement traité ou erroné	Complètement traité et correct	

Le nombre de points accordés par fonctionnalité dépendra de la difficulté de celle-ci.

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.
Procédures	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.
Passage par référence	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 énumérés	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