Traduction des langages

Réalisation d'un compilateur pour le langage RAT

Préambule

— Les TPs sont à réaliser en binôme. Le projet consistera à enrichir le langage RAT, il s'appuiera donc sur le travail réalisé en séance de TPs. Les binômes de projet seront donc les binômes de TP.
— Les TPs s'enchaînent sans que des corrections des étapes intermédiaires ne soient données. Il faut donc, au besoin, finir le travail entre deux séances de TP.
— Un style de programmation fonctionnelle PURE est demandé. Les seuls effets de bord autorisés sont ceux sur les informations associées à un identificateur.
ATTENTION : l'ordre d'évaluation d'Ocaml n'est pas intuitif (dernier paramètre d'un n-uplet en premier, dernier paramètre d'une fonction en premier, opérande droite puis opérande gauche, ...). Ceci n'a aucun impact en programmation fonctionnelle pure, mais en a dès qu'il y a des effets de bord. S'il y a des effets de bord dans `xxx` et `yyy`, privilégiez une écriture de la forme :

```
let e1  = xxx in
let e2 = yyy in
(e1,e2)
```

où c'est vous qui choisissez l'ordre l'évaluation (ordre des `let ... in`), plutôt que `(xxx,yyy)` qui laisse à OCaml le choix de l'ordre d'évaluation (`yyy` sera évalué en premier).
— Séance 1 : Résolution des identifiants
— Séance 2 : Typage
— Séance 3 : Placement mémoire
— Séances 4 et 5 : Génération de code

1 Grammaire formelle du langage Rat

1. $PROG' \rightarrow PROG$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
4. $PROG \rightarrow\ id\ \ BLOC$
5. $BLOC \rightarrow\ \{\ IS\ \}$
6. $IS \rightarrow\ I\ \ IS_1$
7. $IS \rightarrow$
8. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow\ id\ =\ E\ ;$
10. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
11. $I \rightarrow\ print\ \ E;$
12. $I \rightarrow\ if\ \ E\ \ BLOC_1\ \ else\ \ BLOC_2$
13. $I \rightarrow\ while\ \ E\ \ BLOC$
14. $DP \rightarrow$
15. $DP \rightarrow TYPE\ id\ DP$
16. $TYPE \rightarrow\ bool$
17. $TYPE \rightarrow\ int$
18. $TYPE \rightarrow\ rat$
19. $E \rightarrow call\ id\ (\ CP\ )$
20. $CP \rightarrow$
21. $CP \rightarrow E\ CP$
22. $E \rightarrow\ [\ E\ /\ E\ ]$
23. $E \rightarrow\ num\ \ E$
24. $E \rightarrow\ denom\ \ E$
25. $E \rightarrow\ id$
26. $E \rightarrow\ true$
27. $E \rightarrow\ false$
28. $E \rightarrow\ entier$
29. $E \rightarrow\ (\ E\ +\ \ E\ )$
30. $E \rightarrow\ (\ E\ *\ \ E\ )$
31. $E \rightarrow\ (\ E\ =\ \ E\ )$
32. $E \rightarrow\ (\ E\ <\ \ E\ )$

2 Analyses lexicale et syntaxique et génération de l'arbre abstrait

Les analyses lexicales et syntaxiques, ainsi que la génération de l'arbre abstrait, sont fournies. Elles sont réalisées avec à l'aide de Menhir (analyseur LR).

1. Le fichier `ast.ml` contient :
   — une interface qui donne la structure générale des arbres qui seront parcourus et créés par les différentes passes du compilateur ;
   — une interface d'affichage de ces arbres ;
   — un module `AstSyntax` contenant la définition du type des arbres abstraits après l'analyse lexicale et syntaxique. Un module `PrinterAstSyntax` permettant l'affichage des ast précédents.
   — un module `AstTds` contenant la définition du type des arbres abstraits après la passe de gestion des identifiants vu en TD. Le module d'affichage n'est pas fourni et il n'est pas indispensable de le réaliser.

   Il ouvre le module `Type` pour la définition des types fournis dans le langage. Il sera à compléter pour ajouter les structures des arbres pour les différentes passes. L'implantation d'un afficheur pour chaque type d'arbre est optionnelle.
2. Le fichier `lexer.mll` réalise l'analyse lexicale.
3. Le fichier `parser.mly` réalise l'analyse syntaxique et la construction de l'arbre abstrait.

3 Analyse sémantique : réalisation du compilateur par passes

Lorsque l'analyse syntaxique du fichier est réalisée sans erreur, on obtient l'arbre abstrait. Les traitements sémantiques souhaités seront réalisés par passes successives qui effectueront des vérifications sur la structure du programme (gestion des identifiants et typage), et transformeront l'arbre.

La structure générale du compilateur par passes vous est fournie.

— Le fichier `passe.ml` contient :
   — une interface `Passe` qui spécifie qu'une passe est composée de deux types et d'une fonction d'analyse qui prend une expression du premier type et renvoie une expression du second type ;
   — différents modules, conformes à l'interface `Passe`, qui correspondent à des passes "qui ne font rien" (ils seront nécessaires pour tester votre compilateur au fur et à mesure de l'écriture des passes).
— Le fichier `compilateur.ml` contient :
   — un foncteur `Compilateur`, paramétré par quatre `Passe`, spécifiant qu'un compilateur est l'application successive des quatre passes (gestion des identifiants, typage, placement mémoire et génération de code) ;
   — cinq modules instanciant le foncteur `Compilateur` et permettant de réaliser des tests après l'implantation successive des passes.

4 Implantation des passes

4.1 Passe de résolution des identifiants (1 séance)

L'objectif de la séance est de réaliser la passe de résolution des identifiants.

Fichiers fournis :

— un module `Tds` contenant la structure de données représentant la table des symboles ;
— un fichier `testTDS.ml` contenant un grand nombre de tests unitaires pour valider votre implantation.
— un fichier `passeTdsRat.ml` contenant un module conforme à l'interface `Passe` et contenant le code de résolution des identifiants pour les blocs et les instructions.

Il n'est normalement pas nécessaire de modifier le module `Tds`, mais vous être libre de le faire si vous en ressentez le besoin.

Il est important d'utiliser les exceptions fournies dans `exceptions.ml` pour que les tests fournis dans `testTDS.ml` aient du sens.

Vous devez :

1. Compléter la passe de résolution des identifiants (module `PasseTdsRat` conforme à `Passe` dans le fichier `passeTdsRat.ml`).
2. Procéder par étapes (même si ce n'est pas toujours simple) :
   (a) Propager la TDS jusqu'aux instructions et ajouter les variables et constantes dans la TDS (fourni)
   (b) Tester en affichant la TDS
   (c) Tester en déclarant deux variables de même nom dans le même bloc → erreur
   (d) Tester en déclarant deux variables de même nom dans deux blocs différents → OK
   (e) Tester en affectant une variable déclarée → OK
   (f) Tester en affectant une variable non déclarée → erreur
   (g) Tester en modifiant une constante → erreur
   (h) Tester en modifiant une fonction → erreur
   (i) Propager la TDS jusqu'aux expressions
   (j) Tester en utilisant un identifiant déclaré dans un bloc parent → OK
   (k) Tester en utilisant un identifiant non déclaré → erreur
   (l) Tester en passant un identificateur de fonction en paramètre d'une opération → erreur
   (m) Tester en passant un identificateur de variables ou constante en paramètre d'un `call` → erreur
   (n) Propager la TDS jusqu'aux déclarations de fonctions et ajouter les identificateurs de fonctions dans la TDS
   (o) Tester en affichant la TDS
   (p) Tester en déclarant deux fonctions de même nom → erreur
   (q) Propager la TDS jusqu'aux paramètres de fonctions et ajouter les identificateurs des paramètres dans la TDS
   (r) Tester en affichant la TDS
   (s) Tester en déclarant deux paramètres de même nom pour la même fonction → erreur
   (t) Tester en déclarant deux paramètres de même nom dans deux fonctions différentes → OK
3. Valider votre implantation à l'aide des tests fournis (les décommenter dans `testTDS.ml`).

4.2 Passe de typage (1 séance)

L'objectif de la séance est de réaliser la passe de typage.

Fichiers fournis :

- un module `Type` contenant les fonctions nécessaires à la manipulation des types ;
- un fichier `testType.ml` contenant un grand nombre de tests unitaires pour valider votre implantation.

Il n'est normalement pas nécessaire de modifier le module `Type`, mais vous être libre de le faire si vous en ressentez le besoin.

Il est important d'utiliser les exceptions fournies dans `exceptions.ml` pour que les tests fournis dans `testType.ml` aient du sens.

Vous devez :

1. Compléter le fichier `ast.ml` pour ajouter un module conforme à `Ast` et représentant la structure de l'arbre après la passe de typage (écrire son printer n'est pas nécessaire).
2. Écrire la passe de typage (module `PasseTypeRat` conforme à `Passe`) dans un fichier `passeTypeRat.ml`.
3. Procéder par étapes (même si ce n'est pas toujours simple).
4. Valider votre implantation à l'aide des tests fournis.

4.3 Passe de placement mémoire (1 séance)

L'objectif de la séance est de réaliser la passe de placement mémoire. Les informations de placement sont ajoutées à la TDS.

Vous devez :

1. Compléter le fichier `ast.ml` pour ajouter un module conforme à `Ast` et représentant la structure de l'arbre après la passe de placement mémoire (écrire son printer n'est pas nécessaire).
2. Écrire la passe de placement mémoire (module `PassePlacementRat` conforme à `Passe`) dans un fichier `passePlacementRat.ml`.
3. Procéder par étapes (même si ce n'est pas toujours simple).
4. Valider votre implantation à l'aide des tests fournis.

4.4 Passe de génération du code (2 séances)

L'objectif des deux dernières séances est de réaliser la passe de génération de code. Cette dernière passe produit directement une chaîne de caractères, il est inutile de définir une nouvelle structure d'arbre.

Fichiers fournis :

— un module `Code` contenant des fonctions auxiliaires nécessaires à la génération de code ;
— un fichier `testTam.ml` contenant des tests pour évaluer votre implantation.

Il n'est normalement pas nécessaire de modifier le module `Code`, mais vous être libre de le faire si vous en ressentez le besoin.

Vous devez :

1. Écrire la passe de génération de code (module `PasseCodeRatToTam` conforme à `Passe`) dans un fichier `passeCodeRatToTam.ml`.
2. Procéder par étapes (même si ce n'est pas toujours simple).
3. Tester au fur et à mesure en exécutant le code généré à l'aide d'itam (cf Moodle)
4. Valider votre implantation à l'aide des tests fournis.