Projet de compilation
Travail à remettre le 6 avril 2021

Avertissement :
Ce sujet est long car on vous donne beaucoup de matière. Plus court ce serait beaucoup plus compliqué à réaliser.
Ne vous laissez pas impressionner par sa longueur et son apparente complexité.
Ce que vous avez à faire est juste une proposition d'ajout comme il est dit plus loin.
L'intervention dans le code fourni est donc assez limitée en volume de temps et en volume tout court.
Lisez bien tout avant de commencer

Lionel Clément

version `2021-03-17`

Modalité pour rendre le travail

Déposer dans le Moodle du cours, dans *Rendu mini-projet* :
`https://moodle1.u-bordeaux.fr/course/view.php?id=5229`
— Un fichier d'archive qui contient tout le code (mais aucune bibliothèque ni aucun fichier compilé)
— Si besoin un fichier PDF très court qui contiendra quelques notes à destination du correcteur qu'il devra lire avant d'ouvrir votre projet.

1 Introduction

Le langage dont il est question dans ce projet est inspiré d'un langage classique de programmation impérative orienté objet (C++, Java, C#, etc). Nous l'appellerons « *Léa* » (Langage Élémentaire Algorithmique).

Dans ce mini-projet, première partie individuelle, on se contentera de représenter la syntaxe de l'input et de signaler au développeur toute erreur syntaxique ou une erreur de typage en lui donnant, la ligne et la colonne à l'endroit où le compilateur a détecté l'erreur.

J'insiste fortement sur le fait qu'il n'est pas demandé que le programme produise la moindre sortie autre que les messages d'erreur.

La seconde partie du projet sera donnée dans un deuxième temps et sera réalisée par groupes de 4, il s'agira d'aller plus loin avec cet exercice.

Les caractéristiques du langage `Léa` sont les suivantes :
— Orienté objet à déclaration de classes et d'interfaces
— Statiquement typé
— Explicitement typé
— Manipule quelques types de haut niveau prédéfinis (ensembles, listes, applications, intervales, énumérés, etc).

Les spécifications du langage Léa sont fournies dans le document annexe à celui-ci.

2 Travail à réaliser

Choisir parmi la liste suivante **une** fonctionnalité à ajouter au projet. Certains de ces ajouts proposés sont notés comme plus difficiles, il vaut mieux réussir un ajout facile que ne pas réussir du tout. Il s'agit donc de savoir vos limites et de réaliser le mieux possible l'ajout le plus adapté à vos compétences.

La modification consiste à modifier `parser/Parser.y` et `lexer/Lexer.jflex` et à ajouter un fichier d'input.

— Reconnaître l'ajout syntaxiquement et signaler une éventuelle erreur syntaxique.

— Faire l'analyse de type correspondante en signalant une éventuelle erreur.

1. Structure de contrôle `for(Init ; Test ; Succ) Instr` du langage C, C++ ou Java. On testera que `Test` soit bien de type booléen. Il conviendra de pouvoir déclarer une variable dans la partie `Init` et de pouvoir l'exploiter dans les parties `Succ` et `Instr`. Cette variable n'a de portée que dans ces blocs d'instructions.

   Exemple :

```
class main {
        m: list<string>;

        main(args: list<string>) {
                for (i: string := m.iterator() ; m.hasNext() ; ) {
                        for (j: integer := 0; j < 10 ; j++) {
                                writeln (m.next());
                        }
                }
                return 0;
        }

}
```

2. Expression `"? ... : ..."` inspiré des langages C++ et Java. L'opérateur `? Test ? Expr1 : Expr2` est une expression qui a comme valeur `Expr1` quand `Test` reçoit la valeur `true`, `Expr2` sinon. On testera que `Test` soit bien de type booléen, on testera la compatibilité des types `Expr1` et `Expr2` et on représentera le type de l'expression.

   Exemple :

```
class main {
        main(args: list<string>) {
                i, j: integer;
                i := (j==0) ? 36 : j;
                return 0;
        }

}
```

3. Structure de contrôle `switch(Expr){case CASE1:Instr CASE2:Instr ... CASEk:Instr ... [default: I`
   Nous adopterons une syntaxe inspirée du langage Pascal (sans l'usage de `break`). Il faudra vérifier la compatibilité entre le type des éléments de `Expr` et les différentes valeurs données par `case`. On testera que `Expr` est bien un type compatible avec les expressions désignées par `case`.

   Exemple :

```
class main {
        main(args: list<string>) {
                i: integer;
                switch (i) {
                        case 2: writeln(1);
                        case (2*2): writeln(2);
                        case (2*2*2): writeln(3);
                        default: writeln(">3");
                return 0;
        }

}
```

4. Opérateur sur le type ensemble `set<E>`. Il s'agit d'ajouter l'ensemble des opérations permettant de manipuler un ensemble et d'opérer sur deux ensembles (`isEmpty`, `add`, `remove`, `union`, `intersection`,

etc.). Il faudra vérifier que les éléments d'un ensemble reçoivent bien la fonctionnalité permettant de les comparer deux à deux.

Exemple :

```
class main {
        main(args: list<string>) {
                s1, s2, s3, s4: set<integer>;
                s1.add(1);
                s1.add(3);
                s1.add(5);
                s1.add(7);
                s1.add(11);
                s2.add(3);
                s2.add(6);
                s2.add(9);
                s3 := s1.union(s2);
                s4 := s1.intersection(s2);
                return 0;
        }
}
```

5. Opérateur sur le type `map<K, V>`. Il s'agit d'ajouter l'ensemble des opérations permettant de manipuler une relation et d'opérer sur deux relations (`isEmpty`, `put`, `get`, `remove`, `union`, `intersection`, etc.). Il faudra vérifier que les éléments d'une application reçoivent bien la fonctionnalité permettant de les comparer deux à deux.

   Exemple :

```
class main {
        main(args: list<string>) {
                s1, s2, s3, s4: map<string, integer>;
                s1.put(1, "one");
                s1.put(3, "three");
                s1.put(5, "five");
                s1.put(7, "seven");
                s1.put(11, "eleven");
                s2.put(3, "three");
                s2.put(6, "six");
                s2.put(9, "nine");
                s3 := s1.union(s2);
                s4 := s1.intersection(s2);
                writeln(s3.get(3));
                return 0;
        }
}
```

6. (difficile) Type énuméré. Il s'agit de pouvoir ajouter un type énuméré sous la forme d'une liste d'identifieurs `enum<I1, I2, ... , Ik>` où chaque identifieur sera enregistré comme un type nommé. On complètera le code avec la possibilité d'avoir un élément d'énuméré comme expression et en vérifiant son type.

   Exemple :

```
class main {
        main(args: list<string>) {
                e1: enum<BLANC, NOIR, BLEU, ROUGE, JAUNE>;
                e2: enum<OUI, NON>;
                e1 := NOIR;
                e2 := OUI;
```

```
                return 0;
        }
}
```

7. (difficile) Type classe. Il s'agit de déclarer l'ensemble du type contenant les attributs et les méthodes. Vérifier l'usage de l'opérateur `X.Y` qui permet de trouver l'attribut ou la méthode `Y` de l'objet `X` dont le type est une classe qui permet l'accès à ces champs. Il sera aussi demandé de vérifier la correction du type de l'objet construit avec `new`.
   Exemple :

```
class E {
        private i: integer;
        public E(i : integer) {
                this.i := i;
        }
        public function getI(): integer {
                return i;
        }
}

main(args: list<string>) {
        main(args: list<string>) {
                e: E := new E(36);
                writeln(e.getI());
                return 0;
        }
}
```

8. (plus difficile) Structure de contrôle `foreach(Var ; List) Instr` inspiré de `for(Var : List) Instr` du langage C++ ou Java. Il conviendra de pouvoir déclarer une variable dans la partie `Var` qui sera un itérateur de la liste `List`. C'est-à-dire que cette variable est l'instance d'une classe qui implémente `iterator<T>` et `List` l'instance d'une classe qui implémente `iterable<T>`. Cet objet doit pouvoir être exploité dans la partie `Instr` et n'a de portée que dans ce bloc d'instructions. Les types `list<T>`, `set<T>`, `map<T>` et `range<T>` sont également disponibles.
   Exemple :

```
class main {
        m: set<string>;

        main(args: list<string>) {
                foreach (i: string ; m) {
                        foreach (j: integer ; [0..9]) {
                                writeln (i);
                        }
                }
                return 0;
        }

}
```

3 Éléments fournis

1. Fichier pour la compilation `build.xml`
   Il suffit d'utiliser la commande `ant` pour compiler le tout et de produire le fichier `data/progr-1.output`.
   Les compilations intermédiaires sont les suivantes :

— `ant parser`
  Compilation par **bison** du fichier `parser/Parser.y`
  Ceci produit
  — fr/ubordeaux/deptinfo/compilation/lea/parser/Parser.java
  — fr/ubordeaux/deptinfo/compilation/lea/parser/Parser.output
— `ant lexer`
  Compilation par **jflex** du fichier `lexer/Lexer.jflex`
  Ceci produit
  — fr/ubordeaux/deptinfo/compilation/lea/parser/ParserLexer.java
— `ant bin`
  Ceci produit
  `bin/fr/ubordeaux/deptinfo/compilation/lea/Main.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/environment/Environment.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/environment/EnvironmentException.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/environment/MapEnvironment.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/Parser$Context.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/Parser$Lexer.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/Parser$Location.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/Parser$SymbolKind.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/Parser$YYStack.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/Parser.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/ParserLexer.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/parser/Position.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/type`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/type/TType.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/type/Type.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/type/TypeException.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/type/TypeExpr$1.class`
  `bin/fr/ubordeaux/deptinfo/compilation/lea/type/TypeExpr.class`
— `ant data`
  Ceci exécute la commande suivante :
  ```
  java -classpath bin:lib/jflex-full-1.8.2.jar
     fr.ubordeaux.deptinfo.compilation.lea.Main data/input-1/lea
     > data/output 2> log/error
  ```
  ce qui produit les fichiers suivants :
  `data/output`
  `log/error`
— `ant clean`
  Ceci détruit les fichiers suivants :
  `bin/**`
  `src/**/parser/Parser.java`
  `src/**/parser/ParserLexer.java`
  `src/**/parser/Parser.output`
  `data/output`
  `log/error`

2. Fichier exemple `data/progr-1.lea`.
3. Classe principale
   `fr/ubordeaux/deptinfo/compilation/lea/Main.java`
4. Grammaire `Bison`
   `parser/Parser.y`
5. Analyseur lexical `JFLEX`
   `lexer/Lexer.jflex`
6. Environnements
   `fr/ubordeaux/deptinfo/compilation/lea/environment/Environment.java`

`fr/ubordeaux/deptinfo/compilation/lea/environment/EnvironmentException.java`
`fr/ubordeaux/deptinfo/compilation/lea/environment/MapEnvironment.java`

7. Types
   `fr/ubordeaux/deptinfo/compilation/lea/type/Type.java`
   `fr/ubordeaux/deptinfo/compilation/lea/type/TType.java`
   `fr/ubordeaux/deptinfo/compilation/lea/type/TypeExpr.java`
   `fr/ubordeaux/deptinfo/compilation/lea/type/TypeException.java`

A priori seul les fichiers `parser/Parser.y` et `lexer/Lexer.jflex` doivent être modifiés