TP 2 - grammaire du langage

A. Bonenfant - H. Cassé - C. Maurel
M1 Informatique - université de Toulouse

Les sources du TP obtenues et un texte d'analyse sur l'instruction `IF-ELSE` sont à remettre sur Moodle dans le dépôt TP 2 une fois le sujet du TP terminé avant la date limite de dépôt. Ces sources seront utilisées pour l'évaluation du TP !
Attention : une partie du TP est à réaliser en dehors de la séance de TP.
Pour produire l'archive à déposer, il faut taper :

```
make deliver
```

qui crée un fichier nommé `deliver-DATE.tgz`.

L'objectif de ce TP est d'implanter la grammaire du langage Karel.

Dans le TP précédent, on a déjà ajouté les règles syntaxiques pour les commandes `pickbeeper` and `putbeeper` ainsi qu'une règle permettant de reconnaître un test (direction, mur, beeper à la position courante ou beepers dans le sac). Dans ce TP, nous allons nous intéresser à des instructions plus complexes (sélection, répétition) ainsi qu'aux sous-programmes.
Tout au long de ce TP, nous allons ajouter à la grammaire de Karel de nouvelles règles et il faudra, à chaque fois, recompiler le jeu (appel de la commande `make`) et vérifier qu'il fonctionne sur un programme karel particulier. Pour tester, on utilisera la commande :

```
> ./karel-cc samples/test.karel
```

Avec *test* le nom du test.

Soit une erreur de syntaxe sera levée lors de l'analyse signifiant une erreur de votre part dans la règle ajoutée, soit l'analyse se terminera en affichant quelques quadruplets signifiant que la règle ajoutée est juste[1]. Dans le cas où vous constaterez une erreur, vous pourrez vous aider du fichier parser.output qui contient l'analyse LALR(1) d'`ocamlyacc`. Le format utilisé est décrit en annexe à la fin de ce document.

Le déroulement du sujet est incrémental : vous allez enrichir peu à peu la grammaire du langage. Pour éviter de "casser" l'étape précédente, vous pouvez sauvegarder l'état courant de vos fichiers en tapant la commande `make save`. Pour restaurer vos fichiers, vous pourrez taper la commande `make restore`.

[1] Comme il n'y a pas de quadruplet généré pour ces nouvelles instructions, seuls les quadruplets générés dans le code initial seront visible.

1. L'instruction ITERATE

L'instruction `ITERATE` permet de réaliser la répétition simple d'une liste d'instructions. Elle commence par le mot-clé `ITERATE`, suivi d'un nombre entier et du mot-clé `TIMES`. Ensuite vient une instruction qui sera répétée autant de fois que la valeur du nombre entier.
Dans l'exemple ci-après, l'avancée du robot et la rotation à gauche est répétée 4 fois :

```
ITERATE 4 TIMES
     BEGIN
          move;
          turnleft
     END
```

On notera que l'utilisation du `BEGIN` … `END` peut être nécessaire car le corps de la répétition est composée de plusieurs instructions. Cependant, il n'est pas indispensable si l'instruction répétée est unique. Par exemple, l'exemple ci-dessous fait avancer le robot 3 fois :

```
ITERATE 3 TIMES
     move;
```

⌨ A faire
Ajouter la(les) production(s) nécessaire(s) au support de l'instruction `ITERATE` dans la règle `stmt` de l'analyseur syntaxique. On veillera à ce que les tokens nécessaires soient fournis par l'analyseur lexical.
On pourra tester le programme en utilisant `./karel-cc` avec le fichier `iterate.karel`.

2. Blocs d'instruction

⌨ A faire
Lancez la commande suivante:

```
./karel-cc samples/begin.karel
```

Si vous ne constatez pas d'erreur, bravo ! Vous pouvez passer au paragraphe suivant.
Sinon :

1. Editez le fichier `samples/begin.karel` et comprenez d'où vient l'erreur.
2. Le programme peut paraître étrange mais une telle construction va beaucoup nous simplifier la vie et couvre probablement un aspect de la grammaire de Karel qui n'est pas précisé (cela arrive souvent avec les langages définis de manière informelle).
3. Modifiez votre programme pour supporter `samples/begin.karel` et profitez en pour simplifier votre grammaire[2].

[2] Il y a toujours intérêt à avoir la grammaire la plus simple possible : (a) on diminue les chances de faire un bug, (b) on aura moins de code à écrire lors de la génération des quadruplets.

2. L'instruction WHILE

L'instruction `WHILE` permet de répéter une instruction tant que la condition reste vraie. Les conditions possibles sont fournies par la règle `test` développée dans le TP précédent.
Par exemple, l'instruction `WHILE` ci-dessous fait avancer le robot jusqu'à ce qu'un beeper soit trouvé :

```
WHILE not-next-to-a-beeper DO
      move;
```

Ou l'exemple ci-après fait tourner le robot et prendre le beeper tant qu'il se trouve sur un beeper :

```
WHILE next-to-a-beeper DO
      BEGIN
            pickbeeper;
            move;
      END
```

⌨ A faire
Ajouter la production nécessaire au support de l'instruction `WHILE` dans l'analyseur syntaxique. On veillera à ce que les tokens nécessaires soient fournis par l'analyseur lexical. On pourra tester le programme en utilisant `./karel-cc` avec les fichier `while.karel`.

3. Instruction IF

L'instruction `IF` est particulièrement complexe mais, pour l'instant, nous allons seulement nous intéresser à la forme du `IF` sans `ELSE`. Le `IF` est suivi d'un `test` (tout comme le `while`), du mot-clé `THEN` et d'une instruction qui n'est exécutée que si la condition est vraie.

Par exemple, ci-dessous, le robot ne tournera à gauche que s'il se trouve sur un beeper:

```
IF next-to-a-beeper THEN
      turnleft;
```

⌨ A faire
Réalisez l'instruction `IF` dans l'analyseur syntaxique et l'analyseur lexical et testez le avec le programme `if.karel`.
Une fois ce test passé, on réalisera un test global des instructions combinées entre elles avec le programme `global.karel`.

4. Sous-programme

Le langage supporte de manière très légère la notion de sous-programme : il s'agit seulement d'associer une séquence d'instructions avec un identificateur et de pouvoir retourner au site d'appel une fois la séquence terminée. En utilisant cet identificateur, la séquence est alors appelée comme si elle remplaçait l'identificateur. *Il n'y a pas de notion de*

paramètre. Les sous-programmes peuvent appeler d'autres sous-programmes mais il faut qu'ils soient préalablement déclarés.
Un sous-programme est déclaré avec le mot-clé `DEFINE-NEW-INSTRUCTION` suivi de l'identificateur du sous-programme, du mot clé `AS` et d'une instruction. La déclaration des sous-programmes se placent entre les mots-clés `BEGINNING-OF-PROGRAM` et `BEGINNING-OF-EXECUTION` et on peut bien sûr en mettre autant que nécessaire.
Dans l'exemple ci-dessous, un sous-programme `turnright` est déclaré puis utilisé dans un programme principal.

```
BEGINNING-OF-PROGRAM

     DEFINE-NEW-INSTRUCTION turnright AS
          BEGIN
               turnleft;
               turnleft;
               turnleft
          END

     BEGINNING-OF-EXECUTION
          turnright;
          move;
          turnoff
     END-OF-EXECUTION

END-OF-PROGRAM
```

⌨ A faire
La réalisation des sous-programmes est relativement complexe et nécessite d'aller modifier plusieurs règles.

1. Ajoutez le token `ID` au `parser.mly` qui est un terminal prenant comme valeur sémantique une chaîne de caractère `string`. Ajoutez la reconnaissance de ce token à `lexer.mll`. Un identificateur Karel est comme un identificateur C : la première lettre peut-être une lettre ou un souligné "_" alors que les lettres suivantes peuvent être des lettres, des soulignés ou des chiffres.
   On prendra soin de placer la reconnaissance de ce token après les identifiants des mots-clé : pourquoi ?
2. Réalisez une règle `define_new` dans lequel vous mettrez la définition d'un seul sous-programme.
3. Ajoutez des règles permettant de définir une liste potentiellement vide de définitions de sous-programmes et connectez-les avec la définition du programme.
4. Ajoutez aux instructions l'appel de sous-programme qui prend simplement la forme d'un token d'identificateur.
5. Compilez le tout et testez le avec le programme `subprog.karel`.
6. Pour faire une analyse complète du programme, il faut vérifier que le sous-programme n'existe pas déjà à sa déclaration, l'ajouter à la table des symboles et vérifier qu'il existe lorsqu'il est appelé.
   Pour ce faire, nous allons utiliser les primitives OCAML suivantes :

- `is_defined` *id* - teste si id est déjà dans la table ou non,
- `define` *id ad* - déclare un sous-programme s'appelant id et démarrant à l'adresse *ad* (*ad* sera laissé à 0 pour ce TP).

Et l'exception (`SyntaxError` *texte*) pour renvoyer une erreur, en l'occurrence qu'un sous-programme de même nom existe déjà.

La valeur sémantique associée à un symbole (terminal ou non-terminal) est représentée par `$`*i* avec *i* représentant la position du symbole dans la production. Ainsi, dans la production - `DEFINE_NEW_INSTRUCTION ID AS stmt`, la valeur sémantique de l'identifiant correspond à $2 (`ID` est en 2ème position).

Réalisez les tests d'existence de l'identificateur à la déclaration et à l'appel du sous-programme. Testez les différents cas avec `subprog.karel`, `double.karel` et `notexist.karel`.

5. L'instruction IF avec ELSE

L'instruction `IF-ELSE` est assez simple d'un point de vue syntaxique. Il s'agit d'une forme de IF auquel on ajoute le mot-clé `ELSE` et une instruction. Par exemple, l'instruction ci-dessous fait tourner le robot à gauche s'il se trouve sur un beeper, sinon il continue tout droit.

```
IF next-to-a-beeper THEN
      turnleft
ELSE
      move
```

Cependant, comme vous allez l'expérimenter ci-dessous, cette forme pose de nombreux problèmes dans l'analyse LALR(1) que nous allons devoir traiter:

```
IF next-to-a-beeper THEN
           IF facing-north THEN
                move
      ELSE
           turnleft
```

⌨ A faire

1. Avant de casser votre grammaire, vous pouvez sauvegarder l'état courant de vos fichiers en tapant la commande `make save`. Pour restaurer vos fichiers, vous pourrez taper la commande `make restore`.
2. Ajoutez l'instruction `IF-ELSE` à notre analyseur syntaxique et, si nécessaire, à notre analyseur lexical. Que se passe-t-il à la compilation ?
3. La raison qui provoque une erreur à la génération de l'analyseur syntaxique est assez subtile. Quoi qu'il en soit, `ocamlyacc` nous fournit l'analyse des items dans le fichier `parser.output` et les erreurs trouvées sont résumées à la fin du fichier.
   Ouvrez le fichier `parser.output` et analysez son contenu. Vous devez y trouver :
   - la grammaire de Karel en résumé
   - l'analyse des items

- l'identification des conflits décalage / réduction et réduction / réduction

Trouvez et analysez le conflit induit par le `IF-ELSE`.

4. En fait le problème se résume à choisir à quel `IF` se rapporte un `ELSE` précédé par 2 `IF` (exemple ci-dessus) : on choisira le fait que le `ELSE` s'applique au `IF` précédent le plus proche !

5. Pour régler l'ambiguïté introduite par le `IF-ELSE`, il faut changer la grammaire pour (a) conserver le même langage mais (b) supprimer l'ambiguïté. Cette approche est assez compliquée mais elle nous assure de ne plus avoir de message d'alerte. Ce lien de Wikipédia donne plus de détails sur ce problème.
6. Testez votre compilateur avec le fichier `samples/else.karel`.

Aide

La résolution de ce problème est assez compliquée mais on peut s'aider de la remarque suivante : *quand un `IF` se prolonge avec un `ELSE` alors on est sûr que tout `IF` contenu dans la partie `THEN` est régulier, c'est-à-dire qu'il est forcément accompagné d'un `ELSE` ! C'est à dire qu'on ne peut pas réduire un `IF` sans `ELSE` juste avant un `ELSE`.*

Note

Le langage Karel, comme de nombreux langages de programmation, ont une grammaire directement LALR(1) à 90-95%. Seules quelques constructions, le `IF-ELSE` ici, nécessitent une traitement particulier.

Annexe : fichier parser.output

Le fichier `parser.output` contient le résultat de l'analyse LALR(1) presque comme on l'aurait fait à la main. On trouve d'abord la grammaire ré-écrite sans les actions avec les règles numérotées :

```
  1  prog : BEGIN_PROG initial_goto defines_opt BEGIN_EXEC initial_patch stmts_opt END_EXEC
END_PROG
      ...
  27  simple_stmt : TURN_LEFT
  28              | TURN_OFF
  29              | MOVE
  30              | PICK_BEEPER
  31              | PUT_BEEPER
```

Ensuite, on va trouver les items (appelées `state`) tout comme dans l'analyse LR(1) :

```
state 10
    define : def_header . stmts  (8)

    MOVE  shift 14
    TURN_LEFT  shift 15
    TURN_OFF  shift 16
    ....
    .  error
```

```
stmts  goto 24
stmt  goto 25
simple_stmt  goto 26
...
```

L'item 10 correspond seulement à la règle 8 avec le point placé entre `def_header` et `stmts`. Sur le terminal `MOVE`, on fait décalage (`shift`) du symbole et on passe à l'item 14. On fait de même pour les terminaux `TURN_LEFT`, `TURN_OFF`. Au cas où aucun terminal n'est reconnu, il y a toujours un terminal générique "`.`" qui induit une erreur.
Viennent ensuite les non-terminaux qui font passer dans l'état cible d'un `goto` lorsqu'ils sont en sommet de pile. Par exemple, s'il y a un `stmts` en sommet de pile, on passe dans l'item 24.

En dernière partie, on trouve :

- la liste des erreurs (s'il y en a),
- des statistiques sur la table d'analyse.

```
State 69 contains 1 shift/reduce conflict.

45 terminals, 21 nonterminals
57 grammar rules, 81 states
```

Dans l'exemple ci-dessus, on voit qu'on a 45 terminaux, 21 non-terminaux, 57 règles de grammaire et 81 items. Par contre, l'item 69 contient un conflit décalage / réduction. En visualisant l'item 69, on obtient :

```
69: shift/reduce conflict (shift 76, reduce 21) on ELSE
state 69
    stmt : if_then . ELSE stmt  (17)
    reg_stmt : if_then .  (21)
    reg_stmt : if_then . ELSE reg_stmt  (22)

    ELSE  shift 76
```

En effet, sur le terminal `ELSE`, l'analyse hésite entre réduire la règle 21 et décaler le `ELSE` dans la règle 22. Par défaut, il choisit le décalage vers l'item 76.