

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 le 22/10/18. Ces sources seront utilisées pour l'évaluation du TP ! Attention : une partie du TP est à réaliser en dehors de la séance de TP.

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
```

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 juste¹.

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-dessous, l'avancée du robot et la rotation à gauche est répétée 4 fois :

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

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

On notera que l'utilisation du `BEGIN ... END` peut être nécessaire car le corps de la répétition est composée de plusieurs instructions. 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 `simple_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. 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-dessous fait tourner le robot et prendre le beeper tant qu'il est 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 2^{ème} 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 posent de nombreux d'analyse LALR(1) que nous allons devoir traiter.

■ A faire

1. Ajoutez l'instruction `IF-ELSE` à notre analyseur syntaxique et, si nécessaire, à notre analyseur lexical. Que se passe-t-il à la compilation ?

2. 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`.

3. Afin de rendre les choses plus concrètes, peut-on imaginer un programme minimal où l'ambiguïté détectée se produit ? En fait, nous allons voir qu'il faut faire un choix entre 2 dérivations possibles : on choisira le fait que le `ELSE` s'applique au `IF` précédent le plus proche.

4. Pour régler l'ambiguïté introduite par le `IF-ELSE`, on a plusieurs possibilités (*vous devrez justifier votre choix et pourquoi il marche*) :

- `ocamlyacc` fait un choix entre plusieurs décalages, réductions possibles et s'il fait le bon choix, nous pouvons laisser notre grammaire telle quelle.
- On peut 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.
- Utiliser des directives magiques comme `%left` qui donne la priorité pour `ELSE` du décalage sur la réduction :

```
%left THEN
%left 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 un traitement particulier.

Notes (à lire seulement si vous bloquez sur les questions 3 et 4)

La question (3) est assez complexe. Cependant, vous pouvez vous aider de l'exemple ci-dessous :

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

L'option (2) de la question (4) est très 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` !*