TP 1 - Karel

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

1. Présentation

Karel, et le langage éponyme, est un petit robot utilisé par Richard E. Pattis pour enseigner la programmation dans son livre *Karel the Robot: A Gentle Introduction to the Art of Programming*. Le principe est d'apprendre les bases de la programmation (commande, séquence, condition, répétition, etc) en illustrant l'exécution du programme par l'affichage d'un petit robot qui doit, dans un décor donné, réaliser des missions : se rendre à un point donné, déposer ou ramasser des balises, etc.[1]

L'objet de ces TPs et de ce projet est de réaliser le compilateur du langage Karel[2] et de l'intégrer au sein d'un environnement permettant l'affichage de l'animation du robot et de l'exécution du programme traduit. Ci-dessous est présenté un exemple de petit programme en Karel:

```
BEGINNING-OF-PROGRAM
  BEGINNING-OF-EXECUTION
      move;
      turnleft;
      move;
      turnleft;
      move;
      turnleft;
      move;
      turnleft;
      turnoff
  END-OF-EXECUTION
END-OF-PROGRAM
```

Ce programme, très simple, ordonne au robot d'avancer puis de tourner à gauche. Cette action est répétée 4 fois si bien qu'à la fin, le robot a repris sa position initiale et le programme se termine. Bien sûr, on peut écrire des programmes plus compliqués en Karel mais il représente une bonne base pour débuter.

A faire

1. Téléchargez l'archive `karel.tgz` fournie sur le Moodle,

[1] Pour information, Karel est le prénom de Karel Capek écrivain Tchèque qui a inventé le mot Robot dans la pièce de théâtre *R. U. R.*

[2] La description complète du langage Karel est donnée dans le fichier `karel.txt` de l'archive du projet fournie sur Moodle.

2. Décompactez la - `tar xvfz karel.tgz`
3. Compilez la - `cd karel; make`
4. Lancez l'exécution de notre programme
   `./game samples/around.karel samples/empty.wld`

Les fichiers `.karel` contiennent des programmes en Karel (vous pouvez les visualiser et les éditer) alors que les fichiers `.wld` contiennent la description du monde dans lequel va évoluer Karel.

2. Organisation des fichiers

Avant de commencer à étendre le langage Karel, nous allons nous familiariser avec le code qui vous est donné. Il est conseillé de passer un certain temps à comprendre la structure du programme afin de pouvoir aller plus loin. ***Pour vous aider à organiser votre temps de travail, des temps indicatifs sont donnés sur chacune des parties.***

2.1. Le projet (10 mn)

Le projet se compile en utilisant un Makefile. Vous pouvez l'éditer mais il est déconseillé de le modifier si vous n'avez pas l'habitude des Makefile. A chaque fois que vous ferez une modification, il faudra recompiler avec la commande :

```
make
```

Le projet est découpé en module OCaml avec la structure suivante :

- `common.ml` - définitions utilisées pour l'animation du robot,
- `comp.ml` - contient les primitives pour la génération des quadruplets,
- `game.ml` - contient le programme principal (ouvre la fenêtre, charge la carte, etc),
- `karel.ml` - implante les actions du robot,
- `lexer.mll` - contient l'analyseur lexical,
- `parser.mly` - contient l'analyseur syntaxique,
- `quad.ml` - description des quadruplets et fonction d'affichage,
- `ui.ml` - réalise l'interface graphique,
- `vm.ml` - implante la machine virtuelle capable d'exécuter les quadruplets,
- `wlexer.mll` - analyseur lexical pour lire les fichiers .wld,
- `wparser.mly` - analyseur syntaxique pour lire les fichiers .wld.

Il ne vous est bien sûr pas demandé de lire et de comprendre tous les fichiers de ce projet mais seulement ceux décrits dans les paragraphes suivants.

3. Quadruplets et machine virtuelle

3.1. Les quadruplets (20 mn)

Le fichier `quad.ml` contient un type structure OCaml, `quad`, permettant de décrire les quadruplets. L'identificateur de chaque constructeur, ADD, SUB, MUL, etc, permet d'identifier l'opération. Il peut prendre de 0 à 3 paramètres de type entier permettant de représenter un numéro de variable, un littéral entier ou une adresse de quadruplet. La signification de ces constructeurs est donnée plus loin.

La machine virtuelle définie dans `vm.ml` supporte une infinité de variables mais celles-ci doivent être définies par un numéro entier. Ainsi, il appartiendra à notre compilateur d'associer avec chaque variable une valeur entière et de l'utiliser au moment de la génération des quadruplets.

Dans la description suivante, certains variables spéciales sont utilisées :

- PC - le compteur ordinal, contient l'adresse de quadruplet à exécuter,
- S - pile utilisée pour stocker l'adresse de retour pour réaliser des sous-programmes.

`ADD` (*d*, *a*, *b*) - *d* ← *a* + *b*
affecte à la variable *d* la somme des variables *a* et *b*.

`SUB` (*d*, *a*, *b*) - *d* ← *a* - *b*
affecte à la variable *d* la différence des variables *a* et *b*.

`MUL` (*d*, *a*, *b*) - *d* ← *a* × *b*
affecte à la variable *d* le produit des variables *a* et *b*.

`DIV` (*d*, *a*, *b*) - *d* ← *a* / *b*
affecte à la variable *d* le quotient entier de la variable *a* par la variable *b*.

`SET` (*d*, *a*) - *d* ← *a*
affecte à la variable *d* la valeur contenue dans la variable *a*.

`SETI` (*d*, *a*, *b*) - *d* ← valeur(*a*)
affecte à la variable *d* la valeur *a*.

`GOTO` (*n*) - PC ← *n*
réalise un branchement, l'exécution continue au quadruplet *n*.

`GOTO_EQ` (*n*, *a*, *b*) - si *a* = *b* alors PC ← *n*
réalise le branchement sur le quadruplet *n* si *a* = *b*.

`GOTO_NE` (*n*, *a*, *b*) - si *a* ≠ *b* alors PC ← *n*
réalise le branchement sur le quadruplet *n* si *a* ≠ *b*.

`GOTO_LT` (*n*, *a*, *b*) - si *a* < *b* alors PC ← *n*
réalise le branchement sur le quadruplet *n* si *a* < *b*.

`GOTO_LE` (*n*, *a*, *b*) - si *a* ≤ *b* alors PC ← *n*
réalise le branchement sur le quadruplet *n* si *a* ≤ *b*.

`GOTO_GT` (*n*, *a*, *b*) - si *a* > *b* alors PC ← *n*
réalise le branchement sur le quadruplet *n* si *a* > *b*.

`GOTO_GE` (*n*, *a*, *b*) - si *a* ≥ *b* alors PC ← *n*
réalise le branchement sur le quadruplet *n* si *a* ≥ *b*.

`INVOKE` (*d*, *a*, *b*) - spécial
réalise une sorte d'*appel système* auprès de la machine virtuelle; celui-ci est passé au module réalisant l'application Karel. La signification des paramètre *d*, *a* et *b* est dépendante de l'application sous-jacente et expliquée dans un prochain paragraphe.

`STOP` - arrêt
provoque l'arrêt de la machine virtuelle.

Ce module contient également des fonctions pour afficher les quadruplets, `print` et `print_prog`, qui seront utiles pour déboguer le compilateur.

3.2. Implantation du robot (30 mn)

Le fichier `karel.ml` contient l'implantation du robot Karel. Ce fichier est un peu complexe et nous n'examinerons que certaines définitions utilisées par le quadruplet `INVOKE`. Dans le cadre de l'application Karel, il est convenu que le premier paramètre, d, permet d'encoder l'opération. Les paramètres *a* et *b* peuvent être utilisés par l'opération ou non : on les laissera alors à 0 (équivalent du *nil* vu en cours).

On notera que la machine virtuelle `vm.ml` est indépendante du langage Karel et de son application. Leur interface est constituée d'une fonction qui est appelée chaque fois que le quadruplet `INVOKE` est exécuté. Les fichiers `vm.ml` et `quad.ml` peuvent donc être utilisés pour d'autres applications.

Les commandes supportées par Karel sont les suivantes :

- *d* = `Karel.move` (*a*, *b* non utilisés) - avancer en avant,
- *d* = `Karel.turn_left` (*a*, *b* non utilisés) - tourner à gauche.
- d = `Karel.put_beepr` (*a*, *b* non utilisés) - dépose un beeper à la position courante.
- d = `Karel.pick_beeper` (*a*, *b* non utilisés) - récupère un beeper de la position courante.
- d = `Karel.next_beeper` (*a* = numéro de variable, *b* non utilisé) - s'il y a un beeper à la position courante, renvoie 1 dans la variable de numéro *a*, renvoie 0 sinon.
- d = `Karel.no_next_beeper` (*a* = numéro de variable, *b* non utilisé) - s'il n'y a pas de beeper à la position courante, renvoie 1 dans la variable de numéro *a*, renvoie 0 sinon.

Ainsi, si on veut réaliser l'action de tourner à gauche, on construira le quadruplet :

```
INVOKE (Karel.turn_left, 0, 0)
```

A faire
Le programme `console` permet d'exécuter un programme Karel écrit en quadruplets. On va l'utiliser pour bien comprendre le fonctionnement de la VM. Pour ce faire, vous devez (a) éditer le fichier `console.ml`, (b) remplacer le contenu de la

liste `prog` par votre programme en quadruplets, (c) recompiler avec make le programme et (d) l'exécuter. Il va alors afficher l'exécution du programme : les instructions exécutées et l'état du robot.
Le robot démarre à la position (10,10) et est orienté vers le nord.
Réalisez et testez les programmes suivants :

1. Sans utiliser de boucle, réalisez un programme qui fait avancer le robot 4 fois vers le nord.
2. En utilisant une boucle, réalisez un programme permettant au robot de se déplacer 5 fois vers le sud.
3. Faire un programme où le robot dépose un beeper, avance de 4 positions (sans boucle), se retourne et avance jusqu'à avoir retrouvé un beeper.

4. Analyse lexicale, syntaxique et compilation

4.1. Analyse lexicale (30 mn)

Le fichier `lexel.mll` décrit l'analyseur lexical. Ce fichier contient non seulement du code OCaml mais utilise aussi un langage spécial pour décrire l'analyseur lexical. La première partie, entre **{** … **}** permet de mettre des définitions OCaml. Dans ce cas, nous en profitons pour utiliser le module `Parser` et ainsi avoir accès aux identificateurs des unités lexicales / terminaux définies dans l'analyseur syntaxique.

```
{
open Parser
}
```

Les lignes suivantes permettent de définir et de nommer deux expressions régulières, pour représenter les commentaires et les espaces. On remarquera que les caractères sont entre simples apostrophe (comment en OCaml). Il est possible de définir un ensemble de caractères entre crochets **[** … **]**. La forme **[^** … **]** permet d'indiquer l'ensemble de tous les caractères sauf ceux entre crochet.

```
let comment = '{' [^ '}']* '}'
let space = [' ' '\t' '\n']+
```

Ainsi, l'expression régulière `space` décrit une suite non-nulle de caractères qui peuvent être un espace, une tabulation ou un retour à la ligne. L'expression régulière `comment` commence par une accolade ouvrante, puis est constituée par la répétition de n'importe quel caractère sauf une accolade fermante. Elle est enfin terminée par une accolade fermante.

Les lignes suivantes permettent de décrire les expressions régulières représentant les unités lexicales du langage:

```
rule scan =
parse    "BEGINNING-OF-PROGRAM"        { BEGIN_PROG }
```

L'analyseur lexical s'appelle `scan` et reconnaît comme premier mot la chaîne de caractère `"BEGINNING-OF-PROGRAM"` : à ce moment là, il renvoie l'unité lexicale nommée `BEGIN_PROG` qui est définie dans le module `Parser`.

Il en va de même pour les autres mots-clés du langage Karel (le "|" est utilisé pour séparer chaque définition d'unité lexicale) :

```
  |      "BEGINNING-OF-EXECUTION"        { BEGIN_EXEC }
  |      "END-OF-EXECUTION"                  { END_EXEC }
  |      "END-OF-PROGRAM"                { END_PROG }
  |      "move"                          { MOVE }
  |      "turnleft"                      { TURN_LEFT }
  |      "turnoff"                       { TURN_OFF }
  |      ";"                             { SEMI }
```

Les unités lexicales `BEGIN_EXEC`, `END_EXEC`, `END_PROG`, `MOVE`, `TURN_LEFT`, `TURN_OFF` et `SEMI` sont définies dans le module `Parser`.

Les lignes suivantes sont un peu spéciales :

```
  |      space+                      { scan lexbuf }
  |      comment                          { scan lexbuf }
```

Elles permettent de reconnaître soit un ensemble d'espaces, soit un commentaire. Dans les deux cas, ces unités lexicales peuvent être ignorés et n'intéressent pas le compilateur. Le code en OCaml ne renvoie pas d'unité lexicale et rappelle de manière récursive l'analyseur syntaxique, `scan`, sur l'objet implantant le lecteur de fichier, `lexbuf` (défini par défaut dans la déclaration de l'analyseur lexical, commande `parse`). En bref, ces mots seront bien lus par l'analyseur lexical mais ne parviendront jamais à l'analyse syntaxique et seront donc ignorés.

Enfin, la dernière ligne permet de récupérer n'importe quel caractère, _, non reconnu par les expressions régulières précédente et de lever une exception pour prendre en compte l'erreur:

```
  |      _ as c
        { raise (Common.LexerError
             (Printf.sprintf "unknown character '%c'" c)) }
```

A faire

1. Ajouter la reconnaissance des mots-clé `pickbeeper`, `putbeeper` et `next-to-a-beeper`. Pour leur action, on utilisera `{ scan lexbuf }` en attendant de modifier l'analyseur syntaxique. On recompilera avec `make`.
2. Ajouter la reconnaissance des nombres entiers naturels, constitués d'une suite non-vide de chiffres décimaux. De la même manière, on mettra `{ scan lexbuf }` comme action et on recompilera avec make.

4.2. Analyse syntaxique (30 mn)

Le fichier `parser.mly` décrit un analyseur syntaxique LALR(1) en incluant du code OCaml pour associer des actions aux règles reconnues. Tout comme dans le fichier précédent, la première partie entre **%{** … **%}** permet de fournir des déclarations OCaml :

```
%{
open Quad
```

```
open Comp
open Karel
%}
```

Nous en profitons pour ouvrir les modules `Quad`, `Comp` et `Karel`. Dans ce premier TP, nous allons laisser de côté le contenu entre accolades **{** … **}** qui seront expliqués au prochain TP. En bref, elles représentent les actions à réaliser en OCaml quand une production est reconnue (dans notre cas, il s'agira de la génération de code).

Dans `parser.mly`, nous trouvons tout d'abord la définition des unités lexicales utilisées (aussi appelés symboles terminaux ou token en anglais) :

```
%token BEGIN_PROG
%token BEGIN_EXEC
%token END_EXEC
%token END_PROG
%token BEGIN_PROG
%token BEGIN_EXEC
%token END_EXEC
%token END_PROG
```

Du point de vue de l'analyse syntaxique, ce sont juste des identificateurs qui sont produits par l'analyseur lexical `lexer.mll`.

On définit ensuite le nom de l'axiome, `prog` ici, et le type de valeur qu'il renvoie, type nul, `unit`, ici.

```
%type <unit> prog
%start prog
```

On notera qu'à chaque symbole de la grammaire, terminal ou non-terminal, peut être associée une valeur dite *sémantique*. Comme l'analyse LALR(1) est bottom-up, les symboles sont produits au niveau des feuilles de l'arbre de dérivation. Ces valeurs sont rendues disponibles à chaque production reconnue et permettent de produire une nouvelle valeur dans la production courante. De production production, on finit par construire la valeur finale au niveau de la racine de l'arbre de dérivation. La compilation étant réalisée à la volée, nous n'utiliserons pas de valeur dans l'axiome.

Après un séparateur formé de `%%`, on trouve les règles de grammaires qui se conforment à la syntaxe suivante :

NON-TERMINAL **:** $SYMBOLE_{1,1}$ $SYMBOLE_{1,2}$ … **{** $ACTION_1$ **}**
| $SYMBOLE_{2,1}$ $SYMBOLE_{2,2}$ … **{** $ACTION_2$ **}**
…
;

La syntaxe est assez proche de celle vue en cours. Un non-terminal est formé de plusieurs productions séparées par des "**|**". Chaque production est formée d'une suite de symboles, identificateur de terminal ou de non-terminal terminés par une action OCaml entre **{** … **}**.

Par exemple, la règle de `prog` ne contient qu'une seule production définissant un programme débutant par les terminaux `BEGIN_PROG` puis `BEGIN_EXEC`, suivi par le non-terminal `stmts_opt` et terminé par les terminaux `END_EXEC` puis `END_PROG`:

```
prog:    BEGIN_PROG BEGIN_EXEC stmts_opt END_EXEC END_PROG
            { () }
;
```

On remarquera qu'aucune action n'est réalisée lors de la reconnaissance du non-terminal prog : le résultat renvoyé est `()`.

Dans le désordre, nous pouvons directement aller à la définition des instructions, `stmt`, qui n'est constitué que de `simple_stmt`[3]. Le non-terminal `simple_stmt`, quant à lui, est composée de 3 productions contenant chacune 1 mot-clé. Par la suite, nous définirons des instructions plus complexes :

```
stmt:            simple_stmt       { … }
;

simple_stmt:     TURN_LEFT    { … }
|                TURN_OFF          { … }
|                MOVE         { … }
;
```

Un programme est constitué par une liste d'instructions et nous devons définir cette construction dans notre grammaire en utilisant le non-terminal `stmts` :

```
stmts:       stmt                 { () }
|            stmts SEMI stmt       { () }
;
```

Il s'agit ici d'un non-terminal défini de manière récursive afin de supporter n'importe quelle séquence d'instructions. La première production indique qu'une séquence `stmts` peut être composée d'une seule instruction `stmt` : il s'agit du cas final de notre définition récursive. Ensuite, une séquence `stmts` peut être composé d'une séquence `stmts`, d'un point-virgule, `SEMI`, puis d'une instruction seul `stmt`. On notera qu'on a pris soin de réaliser une récursivité à gauche dans notre production pour favoriser le fonctionnement de l'analyseur LALR(1).

Le dernier non-terminal, `stmts_opt`, est légèrement plus compliqué. Il permet de représenter le fait qu'une séquence d'instructions peut être vide :

```
stmts_opt:       /* empty */       { () }
|                stmts             { () }
;
```

La seconde production indique que `stmts_opt` peut être une séquence d'instruction `stmts`. La première est en réalité une production vide : la chaîne `/* empty */` est une chaîne de commentaire pour l'analyseur syntaxique et sera donc ignorée. Elle est ajoutée ici pour bien souligner le fait que la production est vide et donc accepte le mot vide λ. Donc le terminal `stmts_opt` reconnaît soit le mot vide λ, soit une séquence d'instructions `stmts`.

A faire

[3] Nous verrons plus tard l'utilité d'une telle structure.

1. Ajouter les token `PICK_BEEPER`, `PUT_BEEPER` et `NEXT_TO_A_BEEPER` à `parser.mly`. Modifiez le fichier `lexer.mll` pour qu'il les renvoie et recompile le tout avec make.
2. Ajouter les règles permettant de reconnaître les commands Karel suivantes:
    - `pickbeeper` - prend un beeper disponible à la position courante,
    - `putbeeper` - ajouter un beeper à la position courante.

   On pourra poser comme action l'affichage du mot-clé avec un simple `print_string` afin de s'assurer qu'il sera bien reconnu par l'analyse.
3. Ajoutez le token INT dont la valeur sémantique est de type int dans l'analyseur syntaxique et l'analyseur lexical.[4]
4. On pourra tester que le langage est bien reconnue avec le programme `samples/tp1.karel` (option `-c` de `game`).

5. Extensions

(à terminer hors séance)

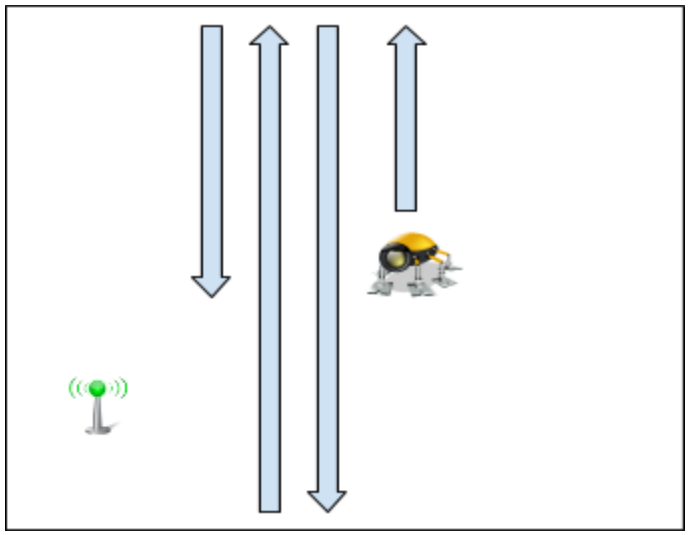
5.1. Ajout sémantique

1. Dans l'analyseur syntaxique, ajoutez les tokens permettant de réaliser les tests:
    - `front-is-clear`, `front-is-blocked`
    - `left-is-clear`, `left-is-blocked`
    - `right-is-clear`, `right-is-blocked`
    - `next-to-a-beeper`, `not-next-to-a-beepe r`
    - `facing-north`, `not-facing-north`
    - `facing-east`, `not-facing-east`
    - `facing-south`, `not-facing-south`
    - `facing-west`, `not-facing-west`
    - `any-beepers-in-beeper-bag` , `no-beepers -in-beeper-bag`
2. Ajoutez une règle (non utilisée pour l'instant) s'appelant `test` qui se dérive en chacun de ces tokens et sera utilisé pour les conditions du `if` et du `while`.
3. Ajoutez la reconnaissance de ces tokens dans le fichier `lexer.mll`.
4. Compilez le tout : un message d'alerte apparaîtra pour dire que la règle `test` n'est pas utilisée. C'est normal et nous l'ignorerons pour l'instant.

5.2. Programmation en quadruplet

En utilisant le programme `console.ml`, écrivez un programme qui fait un parcours vertical depuis sa position courante jusqu'à ce qu'il trouve un beeper. Il partira vers le nord puis, quand il trouvera un mur, il tournera à gauche et avancera avant de repartir vers le sud et ainsi de suite.

[4] La fonction `int_of_string` pourra être utilisée pour convertir le token lu en entier.



On pourra utiliser ce programme sur les cartes suivantes (dont les coordonnées du beeper sont données) :

- `beeper1.wld` - x = 8, y = 8
- `beeper2.wld` - x = 9, y = 8
- `beeper3.wld` - x = 4, y = 4

Pour utiliser ce code, on pourra utiliser `INVOKE` avec les commandes `is_clear` et `is_blocked` qui permettent de savoir, respectivement s'il n'y a pas ou s'il y a un mur dans la direction fournie dans le paramètre *a* (`Karel.left`, `Karel.front`, `Karel.right`) et qui stocke le résultat dans la variable de numéro *b*.

La carte est simplement passée en paramètre à la commande `console` :

```
> ./console samples/beeper1.wld
```