TP 1 - Karel

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

Les sources du TP obtenues sont à remettre sur Moodle dans le dépôt TP 1 avant la date limite de dépôt (voir site Moodle). Ces sources seront utilisées pour l'évaluation du TP !
Attention : une partie du TP est à réaliser en dehors de la séance de TP.

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 et l'animation du robot et 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

[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.

programme se termine. Bien sûr, on peut écrire des programmes plus compliqués en Karel mais celui-ci fournit une bonne base pour débuter.

A faire

1. Téléchargez l'archive `karel.tgz` fournie sur le Moodle,
2. Décompactez la - `tar xvfz karel.tgz`
3. Compilez la - `cd karel; make`
4. Lancez l'exécution de notre programme

```
./karel-cc samples/around.karel
./karel-as samples/around.s
./karel-run samples/around.exe samples/empty.wld
```

Les fichiers suivants ont été utilisés :

- `around.karel` - le programme en Karel (texte),
- `empty.wld` - la description du monde (texte),
- `around.s` - le programme traduit en quadruplet (texte),
- `around.exe` - le programme exécutable prêt à être exécuté.

Les programms suivants (écrits en OCAML) ont également été utilisés:

- `karel-cc` - compilateur de Karel vers les quadruplets,
- `karel-as` - assembleur de quadruplets vers le binaire,
- `karel-run` - machine virtuelle interprétant les quadruplets et produisant l'affichage.

Le langage Karel est un langage impératif possédant les fonctionnalités suivantes :

- commandes du robot : avancer, tourner à gauche, poser / ramasser un marqueur - *beeper*;
- test du robot : détection d'un mur dans une direction donnée, test de direction, test de beeper;
- séquence, sélection, boucle itérant un nombre constant de fois, boucle conditionnelles;
- sous-programme (pas de paramètre).

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 `Makefiles`. A chaque fois que vous ferez une modification des sources, il faudra recompiler avec la commande :

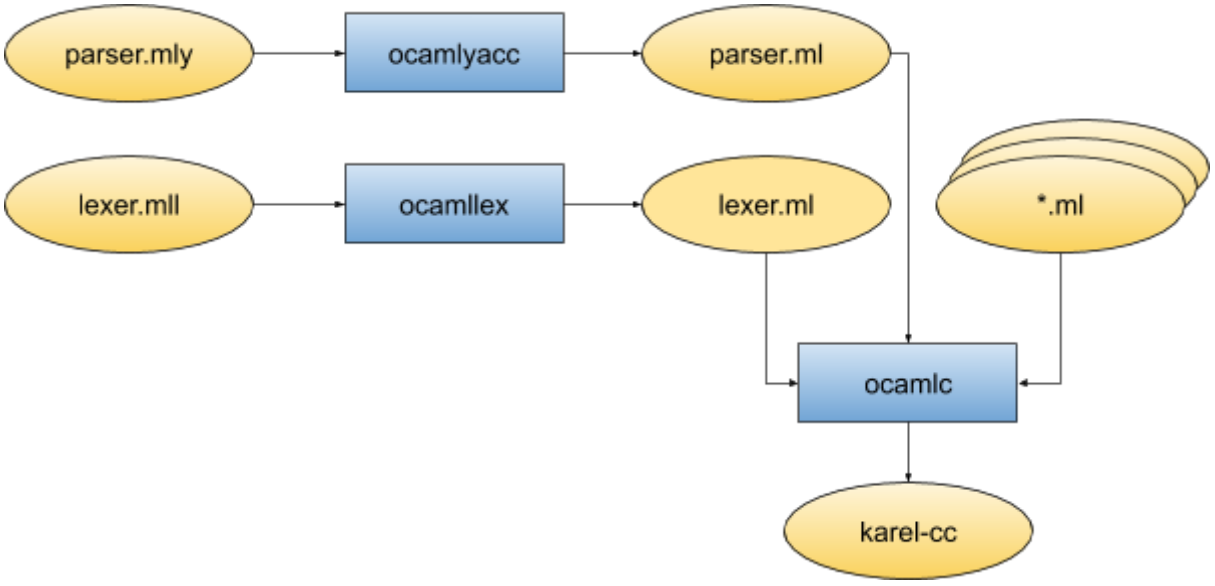
```
make
```

Le projet est découpé en module OCaml qui participent à un ou plusieurs exécutables. Il est utile d'observer les fichiers suivants :

- `quad.ml` - description des quadruplets et fonctions d'affichage,
- `vm.ml` - implante la machine virtuelle capable d'exécuter les quadruplets,
- `comp.ml` - les primitives pour la gestion des quadruplets,
- `lexer.mll` - l'analyseur lexical,
- `parser.mly` - l'analyseur syntaxique.

On notera que la lecture des mondes `.wld` se fait en utilisant un analyseur lexical, `wlexer.mll`, et une analyseur syntaxique, `wparser.mly`.

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.



La figure ci-dessus montre le processus de compilation (réalisé dans le `Makefile`). L'analyseur lexical est décrit dans le fichier `lexer.mll` et convertit en OCAML avec la commande `ocamllex`. L'analyseur syntaxique est décrit dans le fichier `parser.mly` et convertit en OCAML par `ocamlyacc`. On obtient un ensemble de fichiers OCAML qui sont compilés ensemble pour former le compilateur `karel-cc`.

3. Quadruplets et machine virtuelle

3.1. Les quadruplets (20 mn)

Le fichier `quad.ml` contient un type union 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 instructions est donnée plus loin.

La machine virtuelle définie dans `vm.ml` supporte une infinité de variables mais celles-ci doivent être identifiées par un numéro entier. Ainsi, il appartiendra à notre compilateur d'associer à 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,
- M - représente le tableau mémoire contenant les variables.

Dans les listes ci-dessous, pour chaque quadruplet sont données les formes `assembleur` et **`OCAML`** ainsi que la sémantique de du quadruplet séparés par des tirets.

`add` r_d, r_a, r_b - **`ADD`** (*d*, *a*, *b*) - M[*d*] ← M[*a*] + M[*b*]
affecte à la variable *d* la somme des variables *a* et *b*.

`sub` r_d, r_a, r_b - **`SUB`** (*d*, *a*, *b*) - M[*d*] ← M[*a*] - M[*b*]
affecte à la variable *d* la différence des variables *a* et *b*.

`mul` r_d, r_a, r_b - **`MUL`** (*d*, *a*, *b*) - M[*d*] ← M[*a*] × M[*b*]
affecte à la variable *d* le produit des variables *a* et *b*.

`div` r_d, r_a, r_b - **`DIV`** (*d*, *a*, *b*) - M[*d*] ← M[*a*] / M[*b*]
affecte à la variable *d* le quotient entier de la variable *a* par la variable *b*.

`set` r_d, r_a - **`SET`** (*d*, *a*) - M[*d*] ← M[*a*]
affecte à la variable *d* la valeur contenue dans la variable *a*.

`seti` r_d, #*a* - **`SETI`** (*d*, *a*) - M[*d*] ← *a*
affecte à la variable *d* la valeur *a*.

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

`goto_eq` *n*, r_a, r_b - **`GOTO_EQ`** (*n*, *a*, *b*) - si M[*a*] = M[*b*] alors PC ← *n*
réalise le branchement sur le quadruplet numéro *n* si *a* = *b*.

`goto_ne` *n*, r_a, r_b - **`GOTO_NE`** (*n*, *a*, *b*) - si M[*a*] ≠ M[*b*] alors PC ← *n*
réalise le branchement sur le quadruplet numéro *n* si *a* ≠ *b*.

`goto_lt` *n*, r_a, r_b - **`GOTO_LT`** (*n*, *a*, *b*) - si M[*a*] < M[*b*] alors PC ← *n*
réalise le branchement sur le quadruplet numéro *n* si *a* < *b*.

`goto_le` *n*, r_a, r_b - **`GOTO_LE`** (*n*, *a*, *b*) - si M[*a*] ≤ M[*b*] alors PC ← *n*
réalise le branchement sur le quadruplet numéro *n* si *a* ≤ *b*.

`goto_gt` *n*, r_a, r_b - **`GOTO_GT`** (*n*, *a*, *b*) - si M[*a*] > M[*b*] alors PC ← *n*
réalise le branchement sur le quadruplet numéro *n* si *a* > *b*.

`goto_ge` *n*, r_a, r_b - **`GOTO_GE`** (*n*, *a*, *b*) - si M[*a*] ≥ M[*b*] alors PC ← *n*
réalise le branchement sur le quadruplet numéro *n* si *a* ≥ *b*.

`invoke` *d*, *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 au prochain paragraphe.

`stop` - **`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)

Dans le cadre de l'application Karel (instruction `INVOKE`), 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. Ainsi, les appels au quadruplet `INVOKE` sont réalisés par une fonction, propre à l'application, passée en paramètre lors de la création de la machine virtuelle. Les fichiers `vm.ml` et `quad.ml` peuvent donc être utilisés pour d'autres applications.

Les commandes supportées par Karel sont les suivantes :

- *d* = 1 (*a*, *b* non utilisés) - avancer en avant,
- *d* = 2 (*a*, *b* non utilisés) - tourner à gauche.
- *d* = 3 (*a*, *b* non utilisés) - récupère un beeper de la position courante.
- *d* = 4 (*a*, *b* non utilisés) - dépose un beeper à la position courante.
- *d* = 11 (*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* = 12 (*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 (2, 0, 0)
```

A faire

Le programme `karel-exec` permet d'exécuter un programme Karel écrit en quadruplets sans ouvrir de fenêtre mais en affichant les états du robot (position, direction, nombre de beepers). On va l'utiliser pour bien comprendre le fonctionnement de la VM. Pour ce faire, vous devez:

1. éditer un programme en quadruplet, `prog.s` (avec votre éditeur préféré) :
   `geany asm/prog.s`
2. l'assembleur :
   `./karel-as asm/prog.s`
3. l'exécuter :
   `./karel-exec asm/prog.exe`

ou `./karel-run asm/prog.exe`

La syntaxe des quadruplets est très proche de l'assembleur. Les variables / registres ont simplement leur numéro préfixé par "r". Les numéros de quadruplet *n* des instructions de branchement peuvent être remplacé par des étiquettes. Le programme ci-après fait avancer le robot jusqu'à trouver une balise :

```
_start:
    invoke    1, 0, 0
    seti      r0, #0
loop:
    invoke    11, 1, 0
    goto_ne   end, r1, r0
    invoke    1, 0, 0
    goto      loop
end:
    stop
```

Le robot démarre à la position (10, 10) et est orienté vers le nord. Le programme a l'effet suivant :

1. le robot avance (vers le nord)
2. r0 est mis à 0
3. il teste s'il y a un beeper sur la case courante (résultat dans r1)
4. s'il y a un beeper (r1 ≠ r0 ⇔ r1 ≠ 0), il sort de la boucle (étape 6)
5. sinon il avance et recommence à l'étape 3
6. le programme s'arrête.

Réalisez et testez les programmes suivants dans les fichiers correspondants :

1. Sans utiliser de boucle, réalisez un programme qui fait avancer le robot 4 fois vers le nord (dans le fichier `asm/ex1.s`).
2. En utilisant une boucle, réalisez un programme permettant au robot de se déplacer 5 fois vers le sud (dans le fichier `asm/ex2.s`).
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 (dans le fichier `asm/ex3.s`).

4. Analyse lexicale, syntaxique et compilation

4.1. Analyse lexicale (30 mn)

Le fichier `lexer.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. En l'occurrence, 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 (comme en OCaml). Il est possible de définir un ensemble de caractères entre crochets **[** … **]**. Une suite de caractère est obtenue en plaçant "-" entre le plus petit caractère et le plus grande: par exemple, les chiffres sont obtenus par `'0'-'9'`. 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 `parser.mly`.

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 en base 10. De la même manière, on mettra `{ scan lexbuf }` comme action et on recompilera avec make.
3. L'expression régulière reconnue par "_" étant "n'importe quel caractère", observez bien l'usage du "`as c`".

NOTE : pour l'instant, on ne pourra pas tester car il faut aussi modifier l'analyseur syntaxique.

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 : 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* (identificateur, valeur entière, etc). 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 juste au dessus. De production en 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 des grammaires vues 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       { … }
;
```

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

```
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 une séquence formée de plusieurs 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 profiter du 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 un 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

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 recompilez le tout avec `make`.
2. Ajouter les règles permettant de reconnaître les commands Karel suivantes dans la règle `simple_stmt` :
   - `pickbeeper` - prend un beeper disponible à la position courante,
   - `putbeeper` - dépose un beeper à la position courante.

   Afin de les tester, on pourra mettre comme action, entre `{ … }`, 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] Pour déclarer un terminal qui possède une valeur sémantique, on utilisera la syntaxe:

   `%token` ***<type>*** *NOM-TOKEN*

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

Avec *type* étant le type de la valeur sémantique. Comme `ocamlyacc` déclare les terminaux sous forme de type union, on utilisera la syntaxe ci-dessous pour renvoyer le terminal dans `ocamllex` :

| *expression-régulière* **{** *NOM-TOKEN valeur-sémantique* **}**

4. On pourra tester que le langage est bien reconnue avec le programme `tp1.karel` :

```
./karel-cc -c samples/tp1.karel
```

5. Extensions

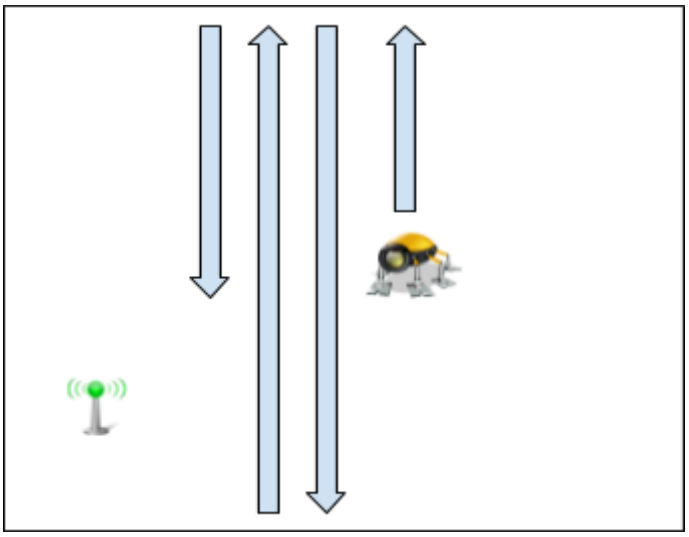
À terminer hors séance.

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-beeper`
   - `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

Ecrivez 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.



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

- `beeper1.wld` - x = 7, y = 7
- `beeper2.wld` - x = 8, y = 7
- `beeper3.wld` - x = 2, y = 2

NOTE : le programme doit être écrit dans le fichier `asm/ex4.s`.

Pour utiliser ce code, on pourra utiliser `INVOKE` avec les commandes `is_clear` (code 5) et `is_blocked` (code 6) 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 (front - 1, left - 2, right - 3) et qui stockent le résultat dans la variable de numéro b.