TP 3 - génération du code

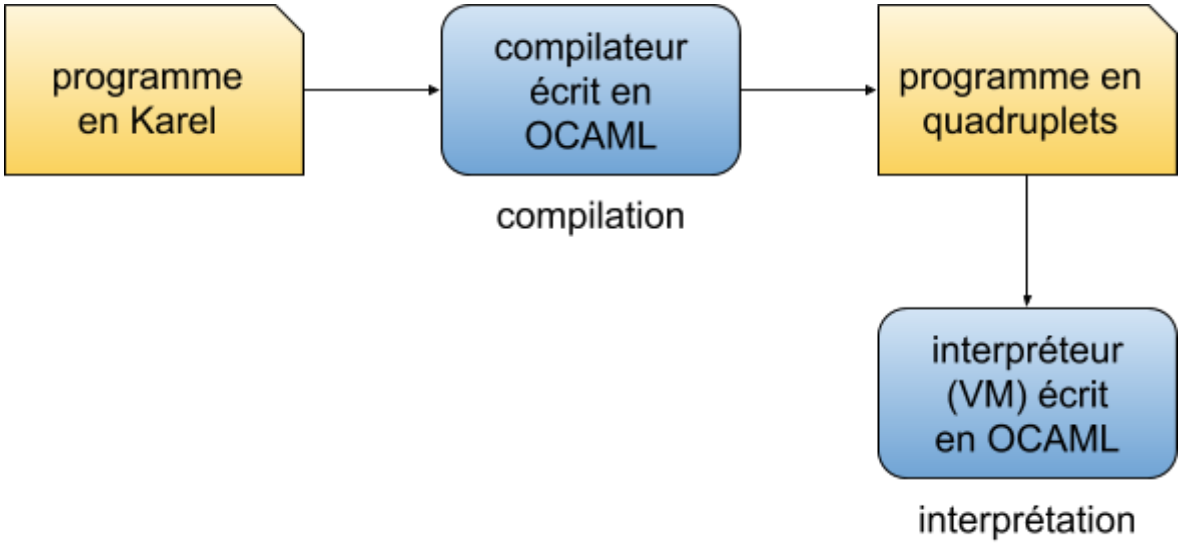
A. Bonenfant - H. Cassé - C. Maurel
M1 Informatique - université de Toulouse

Les sources du TP obtenues sont à remettre sur Moodle dans le dépôt TP 3 une fois le sujet du TP terminé avant le 5/11/18. 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 de générer le code correspondant aux constructions grammaticales développées dans le TP précédent.
Attention ! Lorsqu'on réalise un compilateur, on rappelle qu'on manipule 3 langages :

- Le langage d'entrée sur lequel on réalise l'analyse lexicale et syntaxique - ici, il s'agit de Karel.
- Le langage de sortie vers lequel on veut traduire les programmes du langage d'entrée - ici on a le langage des quadruplets
- Le langage avec lequel on implante le compilateur lui-même - ici, il s'agit d'OCAML.

On ne veut pas exécuter le programme en OCAML : ce seront les quadruplets qui seront exécutés par la VM de Karel!



1. Génération des appels aux primitives

L'objectif de la génération de code est de produire un programme en quadruplets ayant la même sémantique que le programme initial en Karel. Pour ce faire, il faut être capable de manipuler et de stocker les quadruplets ainsi que les définitions de fonction.

Comme il a été vu dans le TP 1, les quadruplets sont représentés par le type OCAML `Quad.quad`. Ces quadruplets sont basiquement des opérations de calcul :

- `ADD` (*d*, *a*, *b*)
- `SUB` (*d*, *a*, *b*)
- `MUL` (*d*, *a*, *b*)
- `DIV` (*d*, *a*, *b*)

Comme leurs noms l'indique, ces opérations permettent d'additionner, de soustraire, de multiplier et de diviser. Leurs paramètres sont de type entier et représentent le numéro de variable/registre à affecter (*d*) ou à utiliser (*a* et *b*). Tout au long de ce sujet, nous nommerons indifféremment variable ou registre les emplacements mémoire dédiés à contenir une valeur entière.

Afin d'être sûr de toujours employer un numéro de variable non-affecté, on pourra utiliser la fonction **`new_temp`** `()` qui renvoie le numéro d'une variable non encore utilisé (on dispose d'une infinité de variables).

L'instruction `SET` (*d*, *a*) permet de copier la valeur de la variable *a* dans la variable *d*. `SETI` (*d*, *i*) permet d'affecter à la variable *d*, la constante *i*. `SETI` est la seule instruction permettant d'affecter une constante *i* dans une variable *d*.

Les quadruplets forment un programme en étant émis séquentiellement avec la fonction **`gen`** *q* avec *q* le quadruplet généré. Cette fonction stocke dans un tableau le quadruplet donné à la suite des quadruplets générés jusque là.

Pour commencer, nous allons générer le code des appels aux primitives du jeu Karel. Il est réalisé par le quadruplet `INVOKE` (*d*, *a*, *b*) qui prend trois paramètres. Le premier paramètre *d* identifie le numéro de la primitive (définie dans le module `Karel`, fichier `karel.ml`) et les paramètres *a* et *b* dépendent du type de la primitive. Les paramètres non-utilisés peuvent être laissés à 0.

Le fichier `parser.mly` donne l'exemple de la génération des instructions `turnleft`, `turnoff` et `move` :

```
simple_stmt:    TURN_LEFT
                  { gen (INVOKE (turn_left, 0, 0)) }
|           TURN_OFF
                  { gen STOP  }
|           MOVE
                  { gen (INVOKE (move, 0, 0)) }
;
```

Pour `turnleft` et `move`, on invoque simplement la primitive du jeu correspondante (les constantes `turn_left` et `move` sont définies dans le fichier `karel.ml`) sans paramètre. Pour `turnoff`, on génère le quadruplet `STOP` qui arrête la machine virtuelle.

A faire

En utilisant `gen` et `INVOKE`, implanter les commandes `pickbeeper` et `putbeeper`. Elles ne prennent pas de paramètres et utilisent les constantes `pick_beeper` et `put_beeper`.

Pour traduire les tests, on va utiliser le quadruplet `INVOKE` avec une des actions *d* de test :

- `is_clear` (*d* = 5) - teste s'il n'y a pas de mur dans la direction *a* qui peut être `front` (*a* = 1), `left` (*a* = 2) ou `right` (*a* = 3) et met le résultat dans la variable dont le numéro est *b*,
- `is_blocked` (*d* = 6) - teste s'il y a un mur dans la direction *a* qui peut-être `front` (*a* = 1), `left` (*a* = 2) ou `right` (*a* = 3) et met le résultat dans la variable dont le numéro est *b*,
- `facing` (*d* = 7) - teste si le robot fait face à la direction `north` (*a* = 1), `east` (*a* = 2), `south` (*a* = 3) ou `west` (*a* = 4) et met le résultat dans la variable dont le numéro est *b*,
- `not_facing` (*d* = 8) - teste si le robot ne fait pas face à la direction `north` (*a* = 1), `east` (*a* = 2), `south` (*a* = 3) ou `west` (*a* = 4) et met le résultat dans la variable dont le numéro est *b*,
- `any_beeper` (*d* = 9) - teste si le robot a encore des beeper dans son panier et met le résultat dans la variable dont le numéro est *a*,
- `no_beeper` (*d* = 10) - teste si le robot n'a plus de beeper dans son panier et met le résultat dans la variable dont le numéro est *a*,
- `next_beeper` (*d* = 11) - teste s'il y a un beeper à la position courante du robot et met le résultat dans la variable dont le numéro est *a*,
- `no_next_beeper` (*d* = 12) - teste s'il n'y a pas de beeper à la position courante du robot et met le résultat dans la variable dont le numéro est *a*.

A la différence des actions `pickbeeper` et `putbeeper`, le test doit s'interfacer avec la règle de plus haut niveau et renvoyer dans quelle variable il stocke le résultat. Cela se fait à travers la valeur sémantique renvoyée par chaque production, résultat de l'évaluation de l'action. Donc les actions de test doivent générer le code en quadruplet et renvoyer le numéro de variable contenant le résultat.

Pour faire la suite, on se rappellera qu'avec une grammaire LR(k), ce sont les actions les plus profondes (feuilles de l'arbre de dérivation) qui sont évaluées avant les actions de règle de plus haut niveau. Par conséquent, s'il y a échange de données entre les règles pour réaliser la traduction, les valeurs sémantiques sont fournies par les sous-règles et associées aux symboles de la production par ***$i***, *i* étant le numéro de symbole dans la production (en commençant à 1). La valeur renvoyée par une règle est la valeur renvoyée par l'expression OCAML et devra être, dans notre cas, le numéro de la variable recevant la valeur résultat, c'est-à-dire *b*.

Pour illustrer l'utilisation des ***$i***, l'exemple ci-dessous montre un exemple de grammaire `ocamlyacc` représentant les expressions et générant des quadruplets correspondant :

```
Expression:
      INT
            { let d = new_temp () in gen (SETI (d, $1)); d }
```

```
|       Expression PLUS Expression
              { let d = new_temp () in gen (ADD (d, $1, $3)); d }
|       Expression MINUS Expression
              { let d = new_temp () in gen (SUB (d, $1, $3)); d }
|       Expression STAR Expression
              { let d = new_temp () in gen (MUL (d, $1, $3)); d }
|       Expression SLASH Expression
              { let d = new_temp () in gen (DIV (d, $1, $3)); d }
|       MINUS Expression
              { let d = new_temp () in
                gen (SETI (d, 0); gen (SUB (d, d, $2)); d }
|       LPAREN Expression RPAREN
              { $2 }
;
```

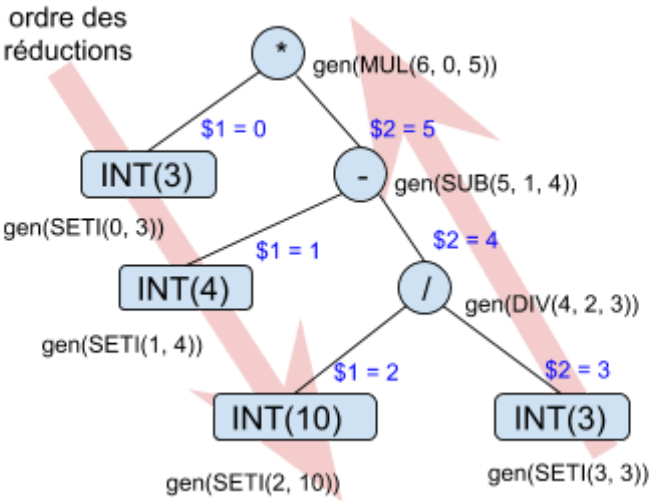
La correspondance entre ***$i*** et symboles est donné pour la production de l'addition (la valeur sémantique de la production, en orange, est le résultat de l'expression OCAML entre accolades) :

Expression: Expression PLUS Expression

$1 **$2** **$3**

{ let d = new_temp () in gen (ADD (d, $1, $3); d }

Ici, chaque traduction renvoie le numéro de la variable *d* qui va contenir le résultat et sera utilisé par les règles de plus haut niveau afin de combiner les opérateurs des expressions. L'expression 3 * (4 - 10 / 3) est analysée et traduite ci-dessous :



Code généré

```
SETI(0, 3)
SETI(1, 4)
SETI(2, 10)
SETI(3, 3)
DIV(4, 2, 3)
SUB(5, 1, 4)
MUL(6, 0, 5)
```

⌨ **A faire**

Faire la traduction correspondant aux productions du non-terminal `test` et vérifiez que votre code compile. Pour l'instant, il est difficile de tester le code produit car il n'y a aucune génération de code pour les instructions structurées (`WHILE`, `ITERATE`, `IF`) : nous testerons l'ensemble dans la section suivante.

2. Génération des instructions structurées

Il faut désormais faire la traduction des instructions structurées comme `ITERATE`, `WHILE`, `IF` et `IF-ELSE`. Ces instructions vont demander l'utilisation des quadruplets de branchements.

Le quadruplet `GOTO` *adr* permet de réaliser un branchement à l'instruction dont l'adresse est *adr*. Lors de la génération du code, les quadruplets sont générés par la fonction **`gen`** et stockés séquentiellement dans un tableau. L'indice du quadruplet dans le tableau est appelé son adresse. Pour obtenir l'adresse du quadruplet qui suit le dernier quadruplet généré, on utilisera la fonction **`nextquad`** `()`. Logiquement, l'adresse obtenue alors pourra être utilisée dans une instruction `GOTO` comme paramètre *adr*.

Il existe des alternatives conditionnelles au `GOTO`, `GOTO_`*cond* (*adr*, *a*, *b*). Selon le résultat de la comparaison de *a* avec *b* selon la condition *cond*, le branchement est réalisé sur adr (cas vrai) ou l'exécution continue en séquence (cas faux). Les conditions peuvent être:

- `EQ` - a = b (égalité),
- `NE` - a ≠ b (différence),
- `LT` - *a* < *b* (plus petit),
- `LE` - *a* ≤ *b* (plus petit ou égal),
- `GT` - *a* > *b* (plus grand),
- `GE` - *a* ≥ *b* (plus grand ou égal).

Avant de faire la traduction de chacune des instructions, il faudra réaliser les étapes suivantes :

- ✎ Traduire un exemple de l'instruction donnée.
- ✎ Proposer un schéma de traduction correspondant.
- ✎ Identifier les difficultés incluant majoritairement :
    - les backpatchs à faire lorsqu'un branchement en avant est réalisé,
    - la transmission par valeur sémantique d'une adresse lorsqu'un branchement en arrière doit être réalisé,
    - la transmission des valeurs sémantiques des sous-production vers les productions typiquement pour passer les numéros de variable contenant le résultat de la sous-production

  On pourra faire des schémas montrant comment les règles s'enchaînent, des sous-productions vers l'axiome.
- ⌨ Réaliser la traduction dans le fichier `parser.mly` qui peut nécessiter de changer légèrement la grammaire du langage pour ajouter des marqueurs ou des supports d'en-tête.

Par exemple, pour le IF seul, on aura les étapes suivantes :

Etape 1 - exemple

IF next-to-a-beeper THEN
mov

qui se traduit en

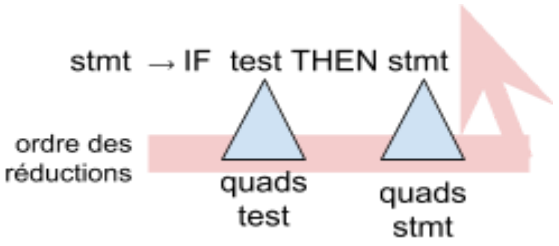
```
            INVOKE (next_beeper, v1, 0)
            SETI (v2, 0)
            GOTO_EQ (etiq, v1, v2)
            INVOKE (move, 0, 0)
      etiq :
```

Etape 2 - schéma de traduction

La règle est : `stmt` ⟶ `IF test THEN stmt`

↕	*test*	(résultat dans *v*)
	`SETI` (*v'*, 0)	(allocation de la variable *v'*)
	`GOTO_EQ` (*e*, *v*, *v'*)	
↕	*stmt*	
e:		

Etape 3 - identification des difficultés



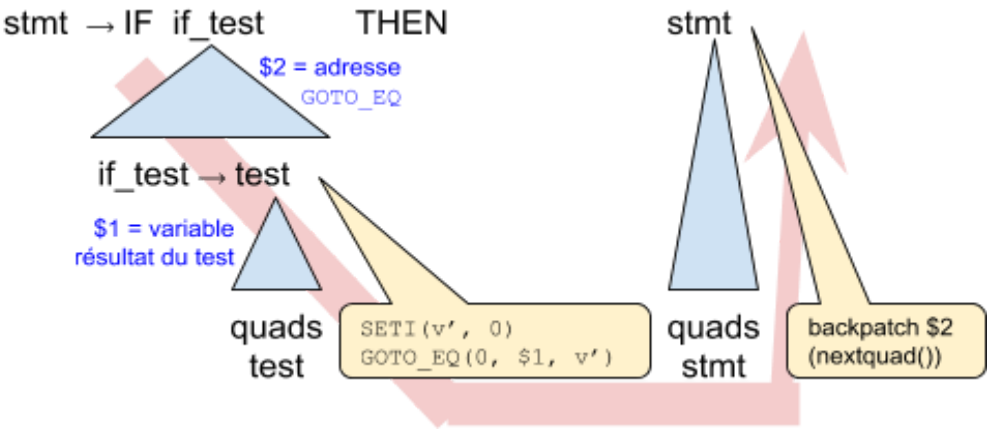
Pour réaliser la traduction, nous devons garder en tête l'ordre de réduction imposé par les grammaire LR : d'abord, les actions des sous-règles sont évaluées de gauche à droite, c'est-à-dire, d'abord `test` puis `stmt`; ensuite l'action de la règle `IF` est appelée.

- Nous voulons insérer du code entre `test` et `stmt` et ce code va utiliser le numéro de variable renvoyé par `test` : mettons `test` dans une sous-production `if_test` où sera réalisée la génération de ce code[1].
- Il faudra ensuite faire un backpatch de l'étiquette *e* vers l'instruction `GOTO_EQ` : il faut donc que `if_test` renvoie l'adresse de l'instruction `GOTO_EQ`.

La fonction OCAML **`backpatch`** adr_{goto} adr_{cible} permet de réaliser un backpatch : elle modifie l'instruction à l'adresse adr_{goto} de manière à ce que sa cible de branchement soit l'instruction à l'adresse adr_{cible}.

Cela donne le schéma de traduction de la page suivante :

[1] On aurait pu mettre la génération de code dans le `test` mais comme le `test` est utilisé ailleurs (`WHILE` par exemple), on pourrait rencontrer des difficultés à harmoniser toutes les utilisations de `test`.



Etape 4 - réalisation de la traduction

```
if_test: test
      {     let v' = new_temp () in
            let _ = gen (SETI (v', 0)) in
            let a = nextquad () in
            let _ = gen (GOTO_EQ (0, $1, v')) in
            a
       }

stmt: …
|     IF if_test THEN stmt
            { backpatch $2 (nextquad ()) }
```

A faire

Réaliser la traduction des instructions :

- `WHILE`
- `ITERATION`
- `IF-ELSE`

On utilisera les fichiers du TP précédent pour vérifier le résultat de la traduction en activant l'option `-d` de la commande `game`. On testera la génération de code de chaque instruction au fur et à mesure.

3. Génération des sous-programmes

(à terminer hors séance)

La génération des sous-programmes est légèrement plus délicate car elle nécessite de générer le code pour le sous-programme et pour l'appel du sous-programme.

3.1. Définition des sous-programmes

En général, le sous-programme est préfixé par du code fixe appelé prologue et terminé par du code fixe appelé épilogue. Le prologue a pour rôle de préparer l'exécution du code du corps du sous-programme comme l'initialisation des variables locales. L'épilogue a pour rôle de nettoyer la mémoire du sous-programme et de réaliser le retour au programme appelant.

Dans le cas de Karel, les sous-programmes sont tellement simples que le prologue est vide. L'épilogue doit seulement réaliser le retour du sous-programme avec le quadruplet `RETURN`. Afin de permettre au sous-programme d'être appelé, il faudra également stocker l'adresse de démarrage du sous-programme : on utilisera la fonction **`define`** *id ad* qui stocke dans la table des symboles l'adresse *ad* de la fonction dont l'identifiant est *id*.

Quadruplet

`RETURN` - PC ← dépiler(S)

réalise un retour de sous-programme en dépilant de S (la pile système) le numéro du quadruplet à exécuter.

Fonction de traduction

`define` *id ad*

Enregistre dans la table des symboles le symbole *id* qui correspond à l'adresse *ad*.

A faire

Réaliser la génération du code pour la déclaration de sous-programme.

3.2. Appel des sous-programmes

L'appel de sous-programme est généralement découpé en 3 phases :

1. calcul des paramètres et, éventuellement, leur empilement ;
2. appel du sous-programme ;
3. nettoyage des ressources utilisées pour stocker les paramètres.

En Karel, il n'y a pas de paramètre. On va seulement réaliser l'étape (2) avec l'instruction `CALL` qui prend comme seul paramètre l'adresse du sous-programme à appeler. Cette adresse peut être obtenue à partir de la table des symboles en utilisant le fonction **`get_define`** *id* avec l'identificateur du sous-programme.

Quadruplet

`CALL` (*n*) - S ←empiler (S, PC + 1); PC ← *n*

réalise un appel au sous-programme dont le premier quadruplet a pour numéro *n*.

Fonction de traduction

`get_define` *id*

Renvoie l'adresse associée à l'identifiant *id* stocké dans la table des symboles.

A faire

Faire la traduction des appels de sous-programme.
Tester avec les programmes Karel du TP précédent.

On peut également tester l'ensemble de la traduction dans une vraie situation :

```
./game samples/maze.karel samples/maze.wld
```

3.3. Programme principal

Si on exécute le programme ainsi obtenu, on va certainement recevoir une erreur disant que la pile est vide ! Cela vient du fait que la machine virtuelle commence à exécuter le code à l'adresse 0 et à l'adresse 0, on va trouver les sous-programmes : on va donc exécuter le premier sous-programme qui va tenter de revenir au programme appelant alors que la pile est vide d'où l'erreur constatée.

A faire

Proposer une solution à ce problème.

Indice : il suffit de mettre comme première instruction un `GOTO` vers la première instruction du programme principal.