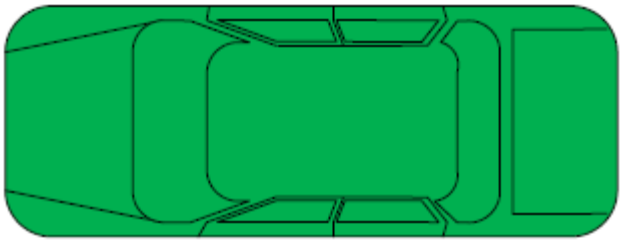
TRUCKY

Le langage des petits camions et des petites voitures



Auteurs : Fabienne Roth et Yann Voumard
Destinataire : Matthieu Amiguet
Section : INF3swe-i
Branche : Langages et compilateurs
Date : 27 mai 2008

haute école neuchâtel berne jura arc ingénierie saint-imier le locle delémont

TABLE DES MATIÈRES

INTRODUCTION

Trucky est un langage de programmation pour les geeks retombés en enfance qui veulent jouer avec des petits camions et des petites voitures. Le principe est le suivant :

1. Ecrire quelques lignes de code qui décrivent les objets et la logique du monde
2. Visualiser le résultat dans une interface graphique lors de l'interprétation

Trucky a été inventé pour le cours de langages et compilateurs à l'école d'ingénieurs ARC. Lors de l'élaboration de ce nouveau langage, les concepteurs se sont inspirés de ce qu'il est possible de faire avec des langages comme NetLogo[1]. Le but principal était de pouvoir faire un maximum de choses en un minimum d'instructions. Les instructions devant être aussi simples que possible.

Ce document s'adresse à des programmeurs qui veulent adapter leurs connaissances aux spécificités de Trucky. Il n'entre pas des détails de fonctionnement et n'explique pas, par exemple, qu'est-ce qu'une condition ou comment utiliser les trois expressions d'une instruction FOR.

PRÉ-REQUIS

Trucky nécessite que les programmes suivant soient installés et fonctionnels :

- Python 2.5 (http://www.python.org)
- pygame 1.8 (http://www.pygame.org)
- Uniquement pour utiliser directement l'outil TY_Parser.py :
    - pydot 0.9.10 (http://dkbza.org/pydot.html)
    - Graphviz 2.16 (http://www.graphviz.org)

[1] http://ccl.northwestern.edu/netlogo

RÉFÉRENCE DU LANGAGE

La syntaxe n'est ni de style C ni de style Python mais un mélange des deux définit comme suit :

- Les retours de ligne séparent les instructions
- Les accolades définissent les blocs
- Les crochets agissent comme des foreachs
- Les points servent d'opérateurs de référence
- Les commentaires sont mono-ligne et commencent par un #

TYPE DE DONNÉES

Le langage définit trois types de données :

1. Les nombres (entiers ou flottants)
2. Les objets (monde, camion ou voiture)
3. Les listes d'objets

VARIABLES

Les variables peuvent être définies à n'importe quel moment et sont toujours liées à un objet étant, par défaut, le monde. Elles peuvent contenir des lettres, des chiffres ou des soulignés (_) mais ne doivent pas commencer par un chiffre.

```
toto = 2 # toto est lié au monde
t = truck()
t.titi = 4 # titi est lié au camion t
```

EXPRESSIONS

Les expressions sont des opérations entre un ou deux opérandes.

```
a += i + 2
p = 10 * (u + 3)
b <= 12
c != h++
b && (i + 2)
```

Les opérateurs supportés sont (par ordre de priorité) :

! ++ --
* / %
+ -
< <= > >=
== !=
&&
||
= += -= *= /= %=

Ceux-ci correspondent aux opérateurs utilisés en langage C et Python. Leur action dépend, naturellement, du type des opérandes. De même, certains ne sont pas supportés par tous les types de données (par ex. : un objet ne peut pas être multiplié par un nombre).

CONDITIONS

Il n'existe qu'une seule structure conditionnelle, les IF.

```
# Simple
if <expression> {
    <statements>
}

# Avec alternative
if <expression> {
    <statements>
} else {
    <statements>
}

# Avec alternatives multiples
if <expression> {
    <statements>
} else if <expression> {
    <statements>
} else {
    <statements>
}
```

BOUCLES

Les boucles disponibles sont : les WHILE, les UNTIL, les FOR et les FOREACH.

```
# While
while <expression> {
    <statements>
}

# Until
until <expression> {
    <statements>
}

# For
for <expression>, <expression>, <expression> {
    <statements>
}

# Foreach
<list> [
    <statements>
]
```

Dans un foreach, l'élément courant est accessible par la variable nommé « current ».

FONCTIONS

rgb(r, g, b)
Crée une *couleur* dont les composantes sont données entre 0 et 255.

truck(x = 0, y = 0, direction = 0)
Crée un camion à la *position* donnée, tourné dans la *direction* spécifiée.
Le camion est automatiquement ajouté à la collection du monde et une référence sur ce nouvel objet est retournée.

car(x = 0, y = 0, direction = 0)
Crée une voiture à la *position* donnée, tournée dans la *direction* spécifiée.
La voiture est automatiquement ajoutée à la collection du monde et une référence sur ce nouvel objet est retournée.

rand(min = None, max = None)
Si *min* et *max* sont spécifiés, retourne un nombre aléatoire entier entre *min* et *max*. (*min* <= x <= *max*)
Sinon, retourne un nombre aléatoire en 0 et 1.

ceil(nb)
Retourne un *nombre* arrondi à l'entier supérieur.

floor(nb)
Retourne un *nombre* arrondi à l'entier inférieur.

paint(delay = 0.5)
Met à jour la représentation graphique du monde et fait une *pause* (en secondes).

echo(what)
Affiche du *texte* dans la console.
A utiliser pour le débogage !

OPÉRATIONS SUR LE MONDE

POSITIONNEMENT ET DIRECTION

Le monde est un plan (2D) dont les deux axes sont découpés en cases. Les déplacements se font dans un nombre limité de 8 directions. Les changements de direction se font par crans (pas) positifs ou négatifs selon le sens de rotation.

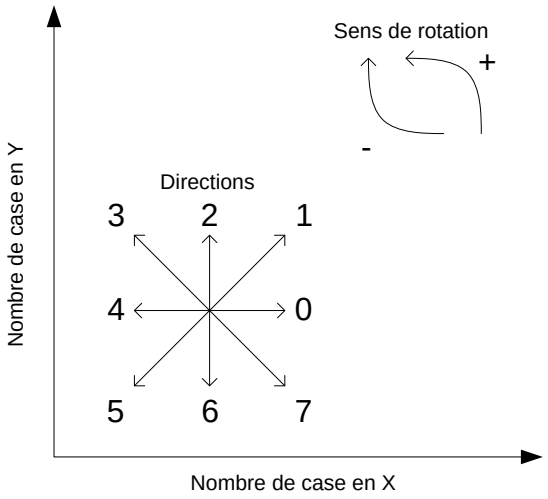


Figure 1 - Logique du monde

WORLD (LE MONDE)

Le monde est une collection étendue. Il possède donc toutes les opérations d'une collection plus celles décrites ci-dessous.

init(nbX = None, nbY = None, color = None)
Initialise le monde (nombre de case en *X*, en *Y* et *couleur* de fond).

getColor()
Retourne la couleur de fond.

setColor(color)
Assigne la *couleur* de fond.

getNbX()
Retourne le nombre de cases en X.

getNbY()
Retourne le nombre de cases en Y.

WORLDOBJECT (LES CAMIONS ET LES VOITURES)

isCar() : number
Retourne 1 si l'objet est une voiture, sinon 0.

isTruck() : number
Retourne 1 si l'objet est un camion, sinon 0.

getX() : number
Retourne la position en *X*.

setX(x) : void
Assigne la position en *X*.

getY() : number
Retourne la position en *Y*.

setY(y) : void
Assigne la position en *Y*.

getDirection() : void
Retourne la direction.

setDirection(direction) : number
Assigne la *direction*.

pick(direction) : collection
Retourne la liste des véhicules qui sont une case devant la case actuelle dans la *direction* donnée.

pickForeward() : collection
Retourne la liste des véhicules qui sont une case devant.

pickBackward() : collection
Retourne la liste des véhicules qui sont une case derrière.

pickNeighbours() : collection
Retourne la liste des véhicules qui sont juste autour de la case actuelle.

pickOver() : collection
Retourne la liste des véhicules qui sont sur la même case.

forward() : void
Avance.

backward() : void
Recule.

turn(delta = 1) : void
Tourne de *delta* cran(s).

COLLECTION (LES LISTES)

add(item) : void
Ajoute un *élément* à la collection.

remove(item) : void
Supprime un *élément* de la collection.

addList(another) : void
Ajoute les éléments d'une *liste* à la collection.

removeList(another) : void
Supprime les éléments d'une *liste* de la collection.

forward(): void
Avance tous les éléments de la collection.

backward(): void
Recule tous les éléments de la collection.

turn(delta) : void
Tourne tous les éléments de la collection de *delta* cran(s).

count() : number
Retourne le nombre d'éléments dans la collection.

cars : collection
Retourne toutes les voitures de la collection.

trucks : collection
Retourne tous les camions de la collection.

all : collection
Retourne tous les éléments de la collection.

UTILISATION DES OUTILS

TY_INTERPRETER.PY

Utilisation : python TY_Interpreter.py <file>

L'interpréteur est l'outil à utiliser pour exécuter (interpréter) les codes écrits en Trucky. Il affiche l'évolution du monde dans une fenêtre PYGAME.

Les voitures sont représentées en vert et les camions en bleu.

Figure 2 - Evolution du monde représentée dans une fenêtre PYGAME

TY_PARSER.PY

Utilisation : python TY_Parser.py <file>

L'analyseur syntaxique affiche l'arbre syntaxique abstrait dans la console et crée un fichier PDF avec une représentation graphique de ce dernier.

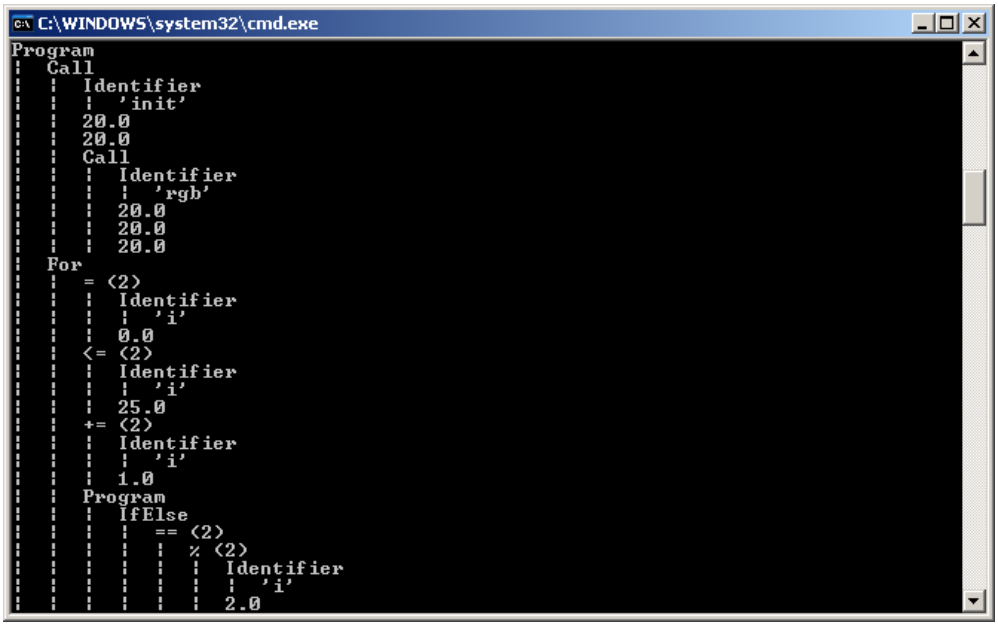


Figure 3 - Arbre syntaxique abstrait représenté dans la console

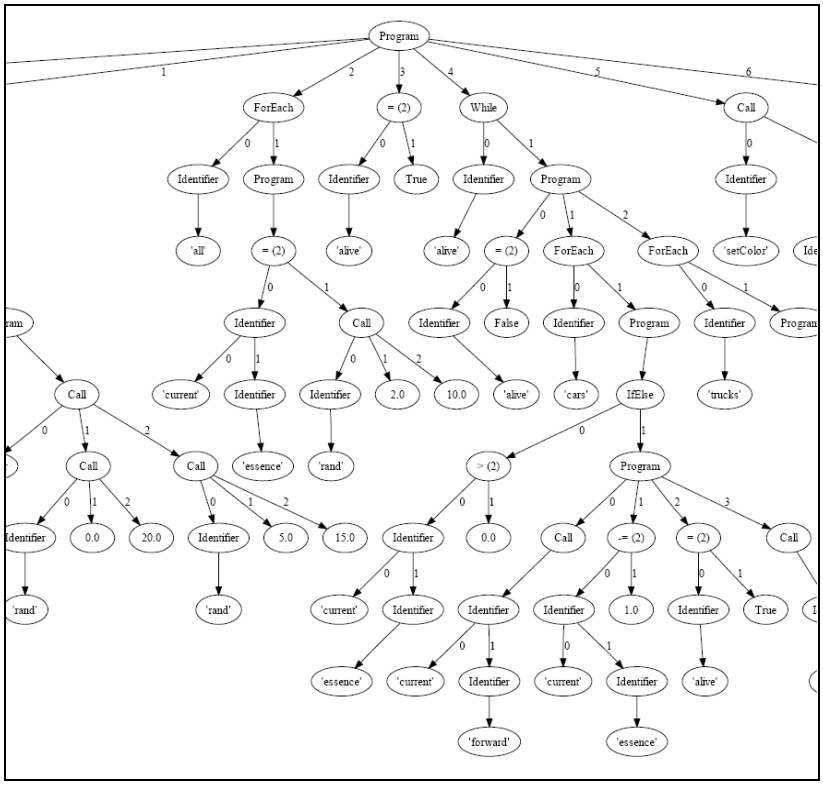


Figure 4 - Arbre syntaxique abstrait représenté graphiquement dans un fichier PDF

TY_LEX.PY

Utilisation : python TY_Lex.py <file>

L'analyseur lexical affiche, dans la console, la suite de lexèmes découverts dans le fichier spécifié.

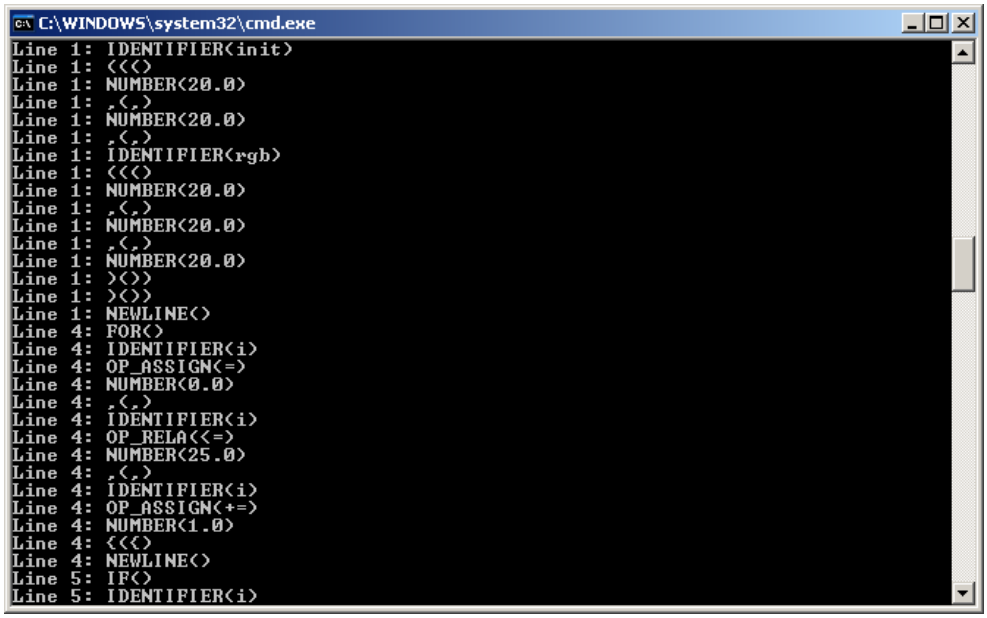


Figure 5 - Liste des lexèmes affichée dans la console

CODES FOURNIS

TESTS

test-lex.txt
Teste tous les lexèmes de l'analyse lexicale.

test-parser.txt
Teste toutes les règles de l'analyse syntaxique.

test-methods.txt
Teste toutes les méthodes du monde.

test-errors.txt
Teste toutes les erreurs pouvant être générées.

EXEMPLES

sample-survivor.txt
Le dernier survivant à gagner !

sample-race.txt
Une course jusqu'à la panne d'essence…

CONCLUSION

La mise au point du langage Trucky a été pour nous l'occasion de mettre en pratique les connaissances acquises durant le cours de langages et compilateurs.

Comme dit précédemment, le langage Trucky se veut simple mais néanmoins puissant. Ainsi, au début de ce projet, nous avons fait certains choix de manière à offrir aux développeurs Trucky le plus de flexibilité possible avec le moins de lignes de code nécessaires. Pour ce faire, nous nous sommes principalement inspirés des langages C et Python afin de tirer le meilleur de chacun d'eux.

Durant la période de développement, nous avons naturellement été confrontés à certains problèmes ayant trait, notamment, à la façon d'implémenter les différentes fonctionnalités que nous voulions offrir avec le langage Trucky. Le principal problème auquel nous avons dû faire face a été la distinction de différents blocs d'instructions. En effet, durant la phase d'analyse, nous avions prévu d'utiliser une syntaxe Python permettant de séparer les blocs d'instructions à l'aide de différents niveaux d'indentation. Cependant, au cours du développement, nous avons constaté que cette manière de faire nous posait un grand problème au niveau de l'analyse lexicale. Nous avions défini que les blocs étaient séparés par des lexèmes de type « INDENT » et « DEDENT » or, en utilisant LEX, il n'est pas possible de générer plusieurs lexèmes en une seule fois. Néanmoins, ceci était nécessaire puisqu'il arrive, bien entendu, qu'il faille, par exemple, « remonter » de plusieurs niveaux d'indentation en même temps et, ainsi, « refermer » plusieurs blocs d'instructions.

Pour faire face à ce problème, nous avons donc choisi d'abandonner cette solution et de revenir à une séparation des blocs d'instructions à la manière du langage C, à savoir, à l'aide d'accolades.

Malgré les problèmes rencontrés, la réalisation de ce projet s'est avérée intéressante et enrichissante. Il ne nous reste plus, désormais, qu'à trouver les futurs développeurs qui prendront du plaisir à utiliser le langage Trucky ! ;-)