Rapport de projet : Honu

Emile Rolley Hugo Thomas

2020/2021

Abstract

Ce document regroupe toutes les informations nécessaires à la compréhension de notre projet réalisé dans le cadre du cours *Grammaires et Analyse Syntaxique*. A savoir, la description des extensions réalisées, les instructions permettant la compilation, l'exécution et l'utilisation de `Honu` et un résumé de notre organisation.

Contents

Présentation

Le projet consiste en l'implémentation d'un interpréteur pour un langage de création d'images basé sur le principe des *turtle graphics*, dans lequel, une suite d'instructions, permettant de déplacer un curseur (ou tortue) sur un canevas, est interprétée pour produire une image.

Nous avons choisi d'aller plus loin et de construire un ensemble d'outils en plus de l'interpréteur demandé. En effet, `Honu` est un langage de programmation strict et dynamiquement typé. Mais également un programme permettant :

- l'interprétation de fichier `.hn` (`Honu`).
- l'utilisation d'un mode interactif pour l'interpréteur.
- la compilation de fichier `.hn` vers différentes cibles.
- la génération de documentation pour les fichier `.hn`.

Le langage comprend également la coloration syntaxique pour l'IDE `Visual Studio Code` et l'éditeur de texte `Vim`.

> **Remarque** : Des exemples d'images produites avec `Honu` sont disponibles dans le fichier *./doc/previews.pdf*.

Organisation

Durant le développement du projet, nous avons suivi la méthode agile *scrum* simplifiée. En effet, travaillant en binôme, nous n'avions pas de *product owner* ni de *scrum master*.

Cependant, nous avons gardé le système de *sprint* et de *product backlog*. Chaque *sprint*, d'une durée de une semaine, se soldait d'une *sprint review* sur `Discord` lors de laquelle nous faisions le point sur les *issues* terminées et celles à faire pour le *sprint* suivant. De plus, nous avons ainsi pu travailler chacun sur toutes les différentes parties du projets.

Hugo c'est occupé de :

- l'implémentation de la première version de l'interpréteur (MR !2).
- l'ajout des expressions arithmétiques dans le langage (MR !3).
- l'ajout de nouvelles instructions dans le langage (MR !6).
- l'ajout de la compilation vers `PNG` et `JPEG` (MR !6).
- l'ajout de l'instruction `if ... then ...` (MR !9).
- l'ajout de nouveaux types (MR !10).
- la résolution de différents *bugs* (MR !14 !15 !17).

Emile quant à lui s'est occupé de :

- mettre en place l'architecture du projet et `GitLab-CI` (MR !1 !4).
- l'ajout des structures de contrôle (MR !5).
- la mise en place de la syntaxe abrégée (MR !7).
- l'ajout des procédures dans le langage (MR !8).
- l'ajout du mode intéractif (MR !11).
- l'ajout de la compilation vers `OCaml` (MR !18).
- l'ajout de la compilation de la documentation (MR !22).
- la résolution de différents *bugs* (MR !12 !13 !16).

La grammaire

Étant donné les extensions qui ont été réalisées, la grammaire initiale a été enrichie (voir Figure 1 et 2). Malgré cela, le langage initial reste reconnaissable.

$S ::= P^*\ D^*\ I$	programme (*axiome*)
$S' ::= P^*\ D^*\ I^*$	programme pour le mode intéractif (*axiome*)
$P ::= (\texttt{Proc} \mid \texttt{proc})\ \ id\ \texttt{(}\ A\ \texttt{)}\ \texttt{=}\ I$	déclaration d'une procédure
$A ::= [\ (id\texttt{,})^*\ id\ ]$	arguments d'une procédure
$D ::= (\texttt{Var} \mid \texttt{var})\ id\ \texttt{;}$	déclaration d'une variable
$I ::= (\texttt{Avance} \mid \texttt{forward})\ E$	fait avancer la tortue
$\mid (\texttt{Tourne} \mid \texttt{turn})\ E$	fait tourner la tortue
$\mid (\texttt{ChangeCouleur} \mid \texttt{color})\ E$	change la couleur du crayon
$\mid (\texttt{ChangeEpaisseur} \mid \texttt{size})\ E$	change la taille du crayon
$\mid (\texttt{Affiche} \mid \texttt{print})\ E$	écrit sur la sortie standard
$\mid (\texttt{Ecrit} \mid \texttt{write})\ E$	écrit sur la fenêtre `Graphics`
$\mid (\texttt{BasPinceau} \mid \texttt{down})$	abaisse le crayon
$\mid (\texttt{HautPinceau} \mid \texttt{down})$	élève le crayon
$\mid (\texttt{Depile} \mid \texttt{pop})$	dépile l'état de la tortue
$\mid (\texttt{Empile} \mid \texttt{push})$	empile l'état courant de la tortue
$\mid id\ \texttt{=}\ E$	affectation d'une variable
$\mid id\ \texttt{+=}\ E$	affectation après addition
$\mid id\ \texttt{-=}\ E$	affectation après soustraction
$\mid id\ \texttt{(}\ A'\ \texttt{)}$	appel d'une procédure
$\mid (\texttt{Debut} \mid \texttt{\{})\ Y\ (\texttt{Fin} \mid \texttt{\}})$	bloc d'instructions
$\mid (\texttt{Si} \mid \texttt{if})\ E\ (\texttt{Alors} \mid \texttt{then})\ I\ [\ (\texttt{Sinon} \mid \texttt{else})\ I\ ]$	branchement conditionnel
$\mid (\texttt{Tant que} \mid \texttt{while})\ E\ (\texttt{Faire} \mid \texttt{do})\ I$	boucle *while*
$A' ::= [\ (E\texttt{,})^*\ E\ ]$	arguments de l'appel d'une procédure
$Y ::= [\ I\ \texttt{;}\ Y\ ]$	ensemble d'instructions

Figure 1: Première partie de la grammaire.

E ::=	n	entier
	| f	réel
	| s	chaîne de caractères
	| c	couleur (`Graphics.color`)
	| id	nom de variable
	| (`vrai` | `faux` | `true` | `false`)	booléens
	| `(` E `)`	expression parenthésée
	| U E	opérateur unaire
	| E B E	opérateur binaire
U ::=	`-`	opposé
	| (`Non` | `not`)	négation logique
B ::=	`*`	multiplication
	| `/`	division
	| `+`	addition
	| `-`	soustraction
	| `&`	conjonction
	| `|`	disjonction
	| `==`	égalité structurelle
	| `!=`	inégalité structurelle
	| `<`	strictement inférieur
	| `>`	strictement supérieur
	| `<=`	inférieur
	| `>=`	supérieur

Figure 2: Deuxième partie de la grammaire.

Avec les définitions suivantes des jetons :

id = [a-z][a-zA-Z0-9]*
n = [1-9][0-9]* | 0
f = [0-9]+[.][0-9]*
s = "[^ "]*"
c = (noir | black | blanc | white | rouge | red | vert | green | bleu | blue | jaune | yellow | magenta | cyan)

> **Remarque** : Toutes les lignes commençant par le caractères `#` sont ignorées.

Les fonctionnalités

En plus des fonctionnalités de bases (sections 1 à 3 du sujet), nous avons ajouté de nombreuses extensions.

> **Remarque** : pour l'implémentation de la boucle `while`, nous avons choisi de limiter le nombre d'itérations à 100 avant de considérer que la boucle est infinie et d'afficher un message erreur. De plus, si l'expression de la condition contient uniquement des consantes et qu'elle ne s'évalue pas à 0, la boucle est également considérée comme infinie.

Extension du langage de base

Comme suggéré dans la quatrième section du sujet, nous avons étendu le langage de base avec de nouvelles instructions comme `ChangeCouleur` ou `ChangeEpaisseur` ainsi que le branchement conditionnel `Si ... Alors ...`. Mais nous avons voulu aller plus loin en rajoutant :

- une syntaxe abrégée
- des instructions supplémentaires
- des opérateurs supplémentaires
- des types
- des procédures

Syntaxe abrégée

Pour faciliter la rédaction et la lisibilité des fichiers de tests, nous avons choisi d'ajouter une syntaxe alternative abrégée (voir La grammaire).

Instructions supplémentaires

En plus des instructions du langage de base, nous avons ajouté :

- `Ecrit | write` qui permet d'écrire sur le canevas d'interprétation.
- `Empile | push` qui permet d'empiler l'état courant de la tortue.
- `Depile | pop` qui permet de dépiler l'état courant de la tortue.
- `Affiche | print` qui permet d'écrire dans la console, et permettre ainsi de faciliter le *debuggage*.
- `+=` qui est un sucre syntaxique pour l'addition suivie de l'assignation d'une variable.
- `-=` qui est un sucre syntaxique pour la soustraction suivie de l'assignation d'une variable.

Opérateurs supplémentaires

Nous avons également ajouté des opérateurs supplémentaires :

- `Non | not` la négation booléenne.
- `&` la conjonction booléenne.
- `|` la disjonction booléenne.
- `==` l'égalité structurelle.

- `!=` l'inégalité structurelle.
- `<`, `>`, `<=`, `>=` les comparateurs usuels sur les entiers et réels.

Procédures

Afin de permettre la factorisation du code nous avons ajouté la possibilité de déclarer des procédures avant les déclarations de variables.

Exemple d'une déclaration de procédure : `proc procName(arg1, arg2, arg3) = { }`

Une procédure est définie par un identifiant, une liste de paramètres et une instruction qui représente le corps de la procédure.

Le langage ne supporte pas l'application partielle, par conséquent une procédure doit être appelée avec le nombre exact de paramètres. Sinon le message d'erreur suivant est affiché : `The procedure '<id>' expects <nb> arguments got <nb>`.

Lors de l'évaluation d'un appel de procédure, la liste d'association d'un paramètre au résultat de l'évaluation d'une expression remplace les variables d'environnement le temps d'évaluer le corps de la procédure.

Ainsi dans le corps d'une procédure, seuls les paramètres sont accessibles et se comportent comme des variables.

Typage dynamique

Les variables peuvent contenir différents types de valeurs :

- des entiers
- des réels (la séparation entre la partie entière et décimale se fait à l'aide d'un point)
- des couleurs (correspondant à celles de `Graphics.color`)
- des booléens (`true`, `false`, `vrai`, `faux`)
- des chaînes de caractères délimitées par `"`.

La vérification de la correspondance entre les types des opérandes aux opérateurs se fait lors l'évaluation d'une expression. Il n'y a donc pas de passe de *type checking*.

Lors de l'évaluation, une expression peut contenir uniquement des opérandes du même type à l'exception des entiers et réels qui peuvent être présent conjointement. Sinon, une exception est levée et le message d'erreur suivant est affiché : `Illegal operation: <exp>`.

Mode interactif

Grâce à la sous-commande `repl`, il est possible d'utiliser l'interpréteur dans un mode intéractif.

L'utilisateur peut alors rentrer un programme `Honu` dont l'axiome de départ a été légèrement modifié (voir Figure 1) afin de pouvoir déclarer des variables ou des procédures sans avoir à terminer par une instruction comme la grammaire de base l'imposait.

Pour la partie CLI, nous avons utilisé la librairie `lambda-term` mais nous avons gardé le même interpréteur que pour la sous-commande `show` en rajoutant seulement un nouveau point d'entrée pour le *parser* (module `Parsing.Parser`).

Utilisation

Une fois le mode intéractif lancé, il est possible de :

- se déplacer dans le *prompt* avec $\Leftarrow$ et $\Rightarrow$
- se déplacer dans l'historique avec $\Uparrow$ et $\Downarrow$
- appliquer l'*autocompletion* avec `Tab`
- faire une recherche dans l'historique avec `Ctrl-R`

- fermer le mode intéractif avec `Ctrl-D`.

Compilation

`Honu` vient avec un compilateur (sous-commande `compile`), permettant de générer des images au format `PNG` ou `JPEG`, ainsi que du code `OCaml` compilable en code binaire grâce au script `compgraph`.

Vers `.jpg` et `.png`

Les images produites correspondent au contenu de la fenêtre graphique après l'interprétation du programme `.hn` fourni.

Vers `.ml`

Quant aux fichiers `OCaml` produits, ce sont des programmes autonomes comprenant une dépendance avec la librairie `Graphics` nécessaire pour l'affichage du résultat.

Au niveau de l'implémentation, le *parser* (module `Parsing`) reste inchangé mais la traduction est faite grâce au module `To_ocaml`.

Exemple

Par exemple pour le programme `Honu` :

```
var i;
{
    i = 0;
    down;
    while i < 3 do {
        if i == 1 then forward i
        else print i;
        i += 1;
    };
}
```

le programme `OCaml` suivant est produit :

```
open Graphics
(* Program entry point. *)
let () =
  (* Start of generated header. *)
  open_graph " 1000x1000";
  set_window_title "honu";
  set_line_width 1;
  set_color black;
  (* End of generated header. *)

  (* Start of translated Honu program. *)
  print_endline ("[ i ] => 0");
  moveto 400 200;
  lineto 400 201;
  print_endline ("[ i ] => 2");
  (* End of translated Honu program. *)

  (* Start of generated footer. *)
  try
```

```
    match (wait_next_event [ Key_pressed ]).key with
    | _ -> ()
  with
    | Graphics.Graphic_failure _ -> ()
  (* End of generated footer. *)
;;
```

Génération de documentation

Il est possible de générer un fichier de documentation au format `Markdown` à partir des commentaires d'un fichier `.hn`, grâce à la commande `doc`.

La grammaire

Pour ce faire nous avons implémenté un nouveau *parser* (`Hdoc.Parser`) correspondant à la grammaire suivante :

$D ::= (T^*\ P)^*$	documentation (*axiome*)
$T ::=$ `~|` `@note` $text^*$ `EOL`	note
| `~|` `@fixme` $text^*$ `EOL`	ajoute une erreur à corriger
| `~|` `@todo` $text^*$ `EOL`	ajoute une tâche à faire
| `~|` `@param` ($id\ text^*$)* `EOL`	fournit la description d'un argument ou d'une variable
| `~|` $text^*$ `EOL`	simple ligne commentée
$P ::=$ (`Proc` | `proc`) id `(` A `) =`	déclaration d'une procédure
| (`Var` | `var`) id `;`	déclaration d'une variable
$A ::=$ [(id`,`)* id]	arguments d'une procédure

Figure 3: Grammaire de la documentation compilable.

Avec les définitions suivantes :

$$id = \text{[a-z][a-zA-Z0-9]}^*$$
$$text = \text{[\^{} \textbackslash n \textbackslash r]}^*$$

> **Remarque** : Tous les commentaires situés après les déclarations de variables doivent commencer par `#` au lieu de `~|`.

Exemple

Par exemple, à partir du code suivant :

```
~| Set the given color and size.
~|
~| @note See documentation for `color` and `size` for more information.
~|
~| @todo add `size s`.
~| @todo remove `"`.
```

```
~|
~| @param c Is the color to set.
~| @param s Is the size to set.
proc setStyle(c, s) = {
  color "c";
}
```

Le fichier `Markdown` suivant est généré :

```
# `TestDoc` documentation

## `setStyle`

Set the given color and size.

> See documentation for `color` and `size` for more information.

### Parameters

* `c`: Is the color to set.
* `s`: Is the size to set.

```
proc setStyle(c, s)
```

Tasks to do

* [ ] add `size s`.
* [ ] remove `"`.
```

Coloration syntaxique

La coloration syntaxique des fichiers `.hn` est disponible pour l'IDE `Visual Studio Code` et pour l'éditeur de text `Vim`.

Pour `Visual Studio Code`, il suffit d'exécuter la commande `make vscode`.

Pour `Vim`, il faut copier le fichier *./syntax-highlight/vim/honu.vim* dans le dossier *$(VIMCONFIG)/syntax*.

Utilisation

Compilation

Le projet est compilable avec la commande `make`. De plus, il est possible d'exécuter les tests unitaires avec `make tests`.

Exécution

A l'issue de la compilation, un lien symbolique (`./honu`) vers l'exécutable est disponible, et s'utilise de la façon suivante :

```
./honu subcommand [arguments] [flags]
```

Les sous-commandes disponibles sont :

- `help`, qui affiche un message d'aide.

- `repl`, qui lance un mode interactif.
- `show FILENAME`, qui affiche dans une fenêtre le résultat de l'interprétation du fichier `FILENAME`.
- `compile FILEIN FILEOUT`, qui compile un programme `Honu` (`FILEIN`) dans un fichier `FILEOUT` dont le type est inféré à partir l'extension du fichier `FILEOUT`. Les extensions supportées sont : `.ml`, `.jpg` et `.png` (extension par défaut si `FILEOUT` ne contient pas d'extension).
- `doc FILEIN FILEOUT`, qui compile la documentation d'un programme `Honu` (`FILEIN`) dans un fichier `Markdown` (`FILEOUT`).

Les *flags* disponibles sont :

- `--verbose`, qui permet l'affichage d'informations supplémentaires (par exemple l'AST construit).
- `--no-color`, qui permet de désactiver l'affichage en couleur dans le terminal.
- `--no-graphics`, qui permet de désactiver l'ouverture de la fenêtre graphique (uniquement disponible pour la sous-commande `show`).
- `--center`, qui permet d'utiliser une position de départ centrée plutôt que (0,0).

Compiler un programme `Honu` vers un fichier exécutable

Afin de faciliter la compilation des fichiers `OCaml` produits par la sous-commande `compile`, nous avons ajouté le script suivant :

```
./compgraph FILENAME
```

Où, `FILENAME` correspond au nom du fichier `OCaml` sans l'extension `.ml`.

Un fichier executable `FILNAME.exe` est alors généré.