UNIVERSITÉ DE RENNES 1

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

Compte-rendu de projet

Compilateur WHILE vers JavaScript

Cours de Compilation
à l'attention de Mr. Ridoux

Chef de projet : GODBILLOT Faustine
Responsable de test : CHIQUET Basile
GUEYE Mame
LABRUE Gwendal
ROCHAT Nicolas
GAUTRAIN Antoine
JEGO Emmanuel

ESIR2
2017-2018

'J'atteste que ce travail est original, qu'il indique de façon appropriée tous les emprunts, et qu'il fait référence de façon appropriée à chaque source utilisée'

Sommaire

Description technique **2**
- Schéma d'exécution 2
  - Pretty-Printing 2
  - Génération de code intermédiaire 3@ 2
  - Génération de code JavaScript 3
- Architecture logicielle 4
  - Compilateur java 4
  - Serveur et interface web 6
  - Run-time WHILE (libWH) 8

Bilan de gestion de projet **9**
- Étapes du développement 9
- Fonctionnalités non-implémentées 10
- Rapports d'activité individuelle 10
  - Basile Chiquet (responsable de test) 10
  - Antoine Gautrain 10
  - Faustine Godbillot (chef de projet) 10
  - Mame Gueye 11
  - Emmanuel Jégo 11
  - Gwendal Labrue 11
  - Nicolas Rochat 11

Description technique

Schéma d'exécution

L'exécution de notre compilateur se fait en 3 temps principaux :

- Pretty-printing du code WHILE écrit par l'utilisateur
- Génération de code intermédiaire au format 3 adresses
- Génération de code JavaScript (code cible) grâce au code intermédiaire

Pretty-Printing

A partir d'un fichier contenant un programme WHILE grammaticalement correct, le pretty-printer se charge d'en créer un nouveau, contenant ce même programme, mais avec une gestion de l'indentation, des espaces, etc. Ce nouveau fichier est écrit dans un répertoire donné et contient donc un programme totalement similaire, écrit de manière plus jolie et lisible.
Cet outil lit le fichier écrit par l'utilisateur et le parcours (plus précisément, il parcourt l'arbre syntaxique du programme écrit). A chaque rencontre d'une règle grammaticale, une règle de pretty-printing lui est appliquée :

- Saut d'une ligne après un "%"
- Indentation du code à l'intérieur d'un "while", d'un "for", etc.
- etc.

Le résultat de cette série d'applications est écrit dans un nouveau fichier.

Génération de code intermédiaire 3@

Pour traduire plus facilement un langage vers un autre, un langage intermédiaire est généralement utilisé : dans notre cas, le code 3 adresses (abrégé code 3@). Il est totalement indépendant du langage cible (pour nous, le JavaScript), bien que l'on puisse le modéliser de manière à faciliter la troisième étape (voire Génération de code JavaScript).
Le principe est de traduire chaque instruction écrite en WHILE vers une instruction écrite en code 3@.
Une instruction 3@ se compose de la manière suivante :

- La commande : while, for, affectation, cons, hd, etc.
- L'adresse de la variable où le résultat sera enregistré (si nécessaire)
- L'adresse de la variable du premier paramètre (si nécessaire)
- L'adresse de la variable du second paramètre (si nécessaire)

Une instruction 3@ s'écrit de la manière suivante :
<Commande, adresse resultat, adresse 1er paramètre, adresse 2nd paramètre>

Quelques exemples pour illustrer cela :

- <READ,_,0,_>
    - Correspond à "read X"

    - 0 est l'adresse de X
- <WHILE [...],_,1,_>
    - Correspond à "while Y do .... od"
    - 1 correspond à l'adresse de Y
    - [...] correspond à la suite d'instructions effectuées à l'intérieur du while
- <CONS, 0, 1, 2>
    - Correspond à "X := (cons Y Z)"
    - 0 correspond à l'adresse de X, 1 à Y, 2 à Z

La génération de code 3@ se fait d'une manière similaire au pretty-printer : les éléments sont reconnus via pattern-matching et sont associés à un code 3@ correspondant.
Afin de stocker les informations du code, nous avons créé une structure adaptée :

Table des fonctions				
Identifiant de fonction	funcEntry			
	Nombre de paramètres d'entrée	Nombre de paramètres de sortie	Liste d'instructions 3@	Table des variables

La table des fonctions est initialisée au début de la lecture du code, et est mise à jour à chaque ligne de code lue. Par exemple, la lecture de "function f:" crée une nouvelle ligne dans la table des fonctions (nouvelle "funcEntry"), et la lecture de "read X, Y, Z" initialise le nombre de paramètres d'entrée à 3 et ajoute ces 3 variables à la table des variables. Concernant les instructions, par exemple "X := Y or Z" génère une instruction "OR", ajoutée à la liste d'instructions.
Certaines instructions WHILE nécessitent la création d'instructions 3@ intermédiaires. Par exemple, "X := A or B and C" sera traduit de la manière suivante :

- <AND,4,1,2>, avec 4 l'adresse d'un registre temporaire, 1 celle de B et 2 celle de C
- <OR,3,0,4> avec 3 l'adresse de X, 0 celle de A, et 4 celle du registre précédemment créé

La génération du code 3@ nécessite également une table de symboles : ce sont des variables globales, utilisables à n'importe quel moment dans un programme WHILE. Elle contiendra, par exemple, le nom des fonctions déclarées dans le programme, pour qu'elles puissent être appelées depuis d'autres fonctions.

Génération de code JavaScript

Enfin, le code 3@ est traduit en code JavaScript en utilisant le résultat de la génération précédemment décrite.
De la même manière que les générateurs précédents, chaque cas est identifié et traduit en JavaScript, en fonction de la commande contenue dans l'instruction 3@.

Nous avons souhaité créer une architecture de code généré un peu particulière, mais utile pour plusieurs raisons. Le code généré a la forme suivante :

```
var whileFunctions = {
      f1 : {
            code : { var v1 = or(v2,v3); ... }
            metadata : { argsIn : 1, argsOut : 2 }
      }
      f2 : { … }
}
```

On remarque que la totalité de code généré est contenue dans un objet “whileFunctions”. Cela permet, notamment pour les tests dynamiques (via une interface web), d’identifier les fonctions à appeler.
Chaque fonction est composée de 2 objets :

- Code : il s’agit du code JavaScript correspondant au code WHILE écrit par l’utilisateur initialement
- Metadata : il s’agit de données supplémentaires, également utiles pour les tests dynamiques. “argsIn” va par exemple permettre de savoir combien de paramètres doivent être demandés à l’utilisateur.

Afin de pouvoir exécuter le code JavaScript obtenu correctement et avoir un fonctionnement identique à WHILE, nous avons dû implémenter différents éléments dans une bibliothèque libWH.js (cf. Architecture logicielle, Run-time WHILE).

Architecture logicielle

Compilateur java

Pour mettre en place le schéma d’exécution décrit précédemment, nous avons dû implémenter le compilateur en Java.

Grammaire et pretty-printer
Pour implémenter la grammaire demandée nous avons utilisé le framework Java Xtext, permettant, dans un premier temps, de définir toutes les règles grammaticales nécessaires. Elle est ainsi définie dans un fichier “Projet.xtext”.

```
PROGRAM :
    (functions += FUNCTION)*
;

FUNCTION :
    "function" name=SYMBOLE ":" def=DEFINITION
;

DEFINITION :
    ("read" inputs=INPUTS)
    ("%" code=COMMANDS)
    ("%" "write" outputs=OUTPUTS)
;
```

Le pretty-printer quant à lui est implémenté en Xtend, dialecte Java, permettant de mettre en place un compilateur beaucoup plus facilement avec le framework Xtext. Le fichier “ProjectGenerator.xtend” contient l’implémentation de cet outil.
Il lit un fichier avec l'extension .wh (code WHILE non pretty-printé) et écrit un fichier GENERATED.whp (code WHILE pretty-printé).

```
// Pour le type "FUNCTION"
def compile(FUNCTION f) {
    // On affiche le nom de la fonction, puis on compile le contenu de la fonction
    '''
        function «f.name»:
            «f.def.compile»
    '''
// f.def.compile est écrit après une tabulation, ce qui va indenter tout le
// contenu de la fonction
}

// Pour le type "DEFINTION"
def compile(DEFINITION d) {
    // on affiche read input, puis le code intérieur indenté, puis write output
    '''
        read «d.inputs.compile»
        %
            «d.code.compile»
        %
        write «d.outputs.compile»
    '''
}
```

Génération de code 3@
La génération de code 3@ nécessite différents éléments, que nous avons implémentés en Java :

- Une classe mère Instruction.java, définissant la structure d’une instruction 3@
    - Adresse résultat, paramètre droit, gauche
    - Accesseurs et modificateurs
- Une classe pour chaque instruction, héritant de Instruction
    - Redéfinition des attributs et/ou méthodes si nécessaire
    - Définition d’une méthode toString() pour l’affichage du code 3@ généré
- Une classe SymTab.java pour définir la table des symboles
    - Table de correspondance nom de symbole/adresse
- Une classe funcTab.java pour définir la table des fonctions pour un programme
    - Table de correspondance nom de fonction/funcEntry
- Une classe funcEntry.java pour définir une fonction contenue dans un programme
    - Nombre d’entrées et de sorties
    - Liste d’instructions
    - Table des variables (correspondance nom/adresse)
- Une classe CodeGenerator.java, générateur de code 3@
    - Lit un fichier WHILE (.wh ou .whc)
    - Remplit la table de fonctions avec le code lu

```
// Pour le type "FUNCTION"
private void compile(FUNCTION f) {
    //on ajoute le nom de la fonction à la table des symboles
    this.symtab.addFun(f.getName());
    this.compile(f.getDef());
}

// Pour le type "DEFINTION"
private void compile(DEFINITION d) {
    //ajout d'une nouvelle fonction à la table des fonctions
    nouvelleFunc = new funcEntry();
    table.addFunc("f"+num,nouvelleFunc);
    num++;

    this.compile(d.getInputs());
    this.compile(d.getCode(), nouvelleFunc.getCode());
    this.compile(d.getOutputs());
}
```

Génération de code JavaScript

La classe JsGenerator.java implémente la génération de code JavaScript. Cette classe se construit à partir d'une table de fonctions et possède une fonction "racine" qui lance la génération de code JS à partir de celle-ci. Elle commence par construire le squelette de l'objet JavaScript puis appelle la fonction translateFunc qui permet de générer le code d'une fonction particulière. C'est l'utilisation de cette fonction de manière récursive qui nous a permis de gérer les imbrications de code.
TranslateFunc traite chaque instruction séparément et nous nous sommes appuyés sur une table de correspondance JS/While afin de traiter chaque cas.
Enfin, tout le code est écrit dans le fichier GENERATED.js et prêt à l'utilisation.

Le reste des fonctions présentes dans cette classe sont des fonctions de formatage, principalement pour gérer l'affichage des "nil".

Serveur et interface web

Étant donné que nous travaillions vers du Javascript, nous avons souhaité créer un éditeur web et pouvoir exécuter le compilateur depuis un navigateur. La première étape de la compilation passe donc par cette interface web (voire annexes).

Tout d'abord, l'utilisateur clique sur "Modifier" dans la zone n°1 : 1-INPUT, ce qui ouvre une zone de texte dans laquelle il peut écrire le code WHILE qu'il souhaite compiler, et valider.

Lorsque l'utilisateur clique sur "COMPILER", le contenu de la zone texte est envoyé à un serveur local. Celui-ci renvoie le code JavaScript correspondant au code WHILE entré **si celui-ci est valide**. Nous avions plusieurs possibilités pour parvenir à notre but (porter le processus de compilation en javascript pour l'intégrer directement à la page avec une balise <*script*> ou bien que la compilation génère une page HTML valide et l'ouvre par exemple).

Le choix du serveur semblait être le plus abordable en un minimum de temps, étant basé sur Java, le même langage que notre projet. Cela nous permet d'appeler la fonction de compilation sur les requêtes transmises au serveur et de retourner le résultat au format JSON. De plus, on peut distribuer le serveur sous la forme d'un JAR exécutable en console.

Lorsque l'utilisateur appuie sur le bouton COMPILER, une requête asynchrone est envoyée au serveur local à l'adresse **http://localhost:3333/** qui contient le code entré par l'utilisateur (cf FormData).

```
Resource resource = set.getResource(URI.createFileURI("./GENERATED.wh"), true);

//pretty printer
fileAccess.setOutputPath(".");
GeneratorContext context = new GeneratorContext();
context.setCancelIndicator(CancelIndicator.NullImpl);
try{
    generator.generate(resource, fileAccess, context);
}catch (Exception e){
    throw new Exception("Error : incorrect WHILE code !!");
}

//3@ generator
CodeGenerator gen = new CodeGenerator();
try{
    gen.generate(resource);
}catch (Exception e){
    throw new Exception("Error : 3@ code generation failed !!");
}

//js generator
JsGenerator jsGen = new JsGenerator(gen.getFuncTab());
String mainFunction = "";
try{
    mainFunction = jsGen.translate();
}catch (Exception e){
    throw new Exception("Error : JavaScript generation failed !!");
}
return mainFunction;
```

Le serveur reçoit donc suite à cette requête le code en WHILE, qu'il écrit dans un nouveau fichier (*GENERATED.wh*).
Comme on peut le voir ci-dessus, le fichier est tout d'abord chargé depuis son emplacement, puis il est pretty-printé avec la fonction "generate" de **ProjectGenerator** et enregistré dans un nouveau fichier *GENERATED.whp*. Ensuite, le code 3@ est généré avec **CodeGenerator** et stocké dans l'objet gen. Enfin, le code JS est généré avec **JsGenerator** et écrit dans un nouveau fichier *GENERATED.js*.

Le serveur fournit alors comme réponse à la requête (cf paradigme client-serveur) le contenu du fichier *GENERATED.js.*
Cette réponse est contenue dans un objet JSON qui associe en l'occurrence à la clé **code** le contenu de *GENERATED.js*. La clé "err" permet quant à elle de transférer un message d'erreur et de réagir en fonction sur l'interface (affichage d'un message d'erreur).

Côté interface client c'est une fonction de callback qui gère la réponse. Le code est affiché dans la zone n°2 : 2-OUTPUT.
Puis nous avons recours à la fonction **eval()** de JavaScript pour évaluer ce code et ainsi le rendre exécutable.

C'est dans la zone n°3 : 3-CONSOLE, que nous affichons la fonction évaluée suite à la compilation et lançable. C'est ici que la façon de stocker les informations du code est importante : en effet, pour exécuter une fonction, il est nécessaire de l'appeler avec le bon nombre de paramètres. Pour ce faire, nous avons décidé d'associer à chaque fonction compilée un attribut **code** et un attribut **metadata**, mais également fait en sorte que les fonctions JavaScript prennent en paramètre un tableau et retournent également un tableau.

De la sorte, en lisant la valeur **metadata.argsIn,** on peut déduire la taille du tableau à passer en entrée à notre fonction : le tableau est ainsi construit sur mesure selon les métadonnées fournies.
La fonction atteignable via cette troisième zone est la dernière fonction définie dans le programme WHILE.

Pour des informations concernant la méthode à suivre pour tester le compilateur, un fichier **README.txt** se trouve dans le dossier “interface” rendu.

Run-time WHILE (libWH)

Pour rendre le code JavaScript exécutable, il a fallu définir une bibliothèque libWH.js (donc **implémentée en JavaScript**), comme expliqué précédemment.
Elle contient :

- La définition du type Tree, correspondant à l’arbre binaire WHILE
- La constante nil → Tree(null,null)
- Une fonction pour les opérateurs and, or, cons, list, tl, hd, eq
    - (cons X Y) → cons(X,Y)
    - (X or Y) → or(X,Y)
    - (tl X) → tl(X)
    - Etc.
- Une fonction evalT() pour évaluer la valeur booléenne d’un arbre binaire
- Une fonction countIt() pour évaluer le nombre d’itération à effectuer pour une boucle for

Cette bibliothèque est directement liée à la page html de l’interface via la balise suivante :

```
<script type="text/javascript" src="./libWH.js"></script>
```

Bilan de gestion de projet

Étapes du développement

Au début, nous avons mis en place plusieurs outils de gestion de projet et de communication : Github, trello et slack. Très vite, nous nous sommes rendus compte que nous communiquions mieux sur Facebook. Le trello a été utilisé ponctuellement pendant les premières semaines, mais comme la plupart du travail consistait en de la recherche et de la compréhension, il était difficile de répartir des tâches. Afin de pouvoir travailler efficacement depuis chez nous, nous avons détaillé les étapes de mise en place du projet sur la documentation Github, afin que chacun puisse travailler depuis chez lui le plus tôt possible.

Les premières étapes de compréhension de la problématique ont été les plus difficiles. Les avancées se faisaient en groupe, particulièrement lors des présentations ou nos questions pouvaient être résolues. Nous avons commencé à répartir les tâches entre nous lorsque les grandes lignes du projet étaient plus claires et que l'architecture générale du pretty-printer était décidée, mais le sprint était déjà bien avancé et nous avons un peu manqué de temps. Nous avons ensuite rencontré un problème d'ambiguïté sémantique lors de la gestion des expressions. Nous avons essayé d'utiliser des prédicats syntaxiques pour le résoudre, mais cela n'a pas suffit. Nous avons finalement dû résoudre ce problème par de la factorisation dans les règles grammaticales. Les autres tâches de fin de sprint (rédaction du --help, création d'un JAR, création de fichiers de tests) n'ont pas posé de problème. Différents membres de l'équipe les ont pris en charge, tandis que d'autres se concentraient sur la difficulté d'ambiguïté dans les expressions.

Le début du deuxième sprint a soulevé les mêmes difficultés que le premier. Les difficultés de compréhension ont néanmoins été plus vites surmontées avec les présentations. Nous avons vite mis en place une architecture générale du projet, sur laquelle plusieurs personnes ont pu travailler en même temps. Le travail était plus divisé car les tâches étaient plus claires, mais l'utilisation de chacun du repository Github permettait un suivi général du projet. L'interface web, la traduction en code 3@ et la traduction en code JS ont été faits de manière séparée, mais l'efficacité individuelle de chacun a permis d'obtenir un résultat cohérent et fonctionnel.

Etant donné les circonstances, à savoir une période donnée pour réaliser le troisième et dernier sprint, nous l'avons commencé avant d'achever totalement le second. Ce dernier était plus facile à réaliser et nous pouvions commencer à voir le résultat final de notre travail : enfin du code JavaScript ! Nous avons enfin fait la connexion avec notre serveur java pour obtenir une interface fonctionnelle, en tentant de la rendre ludique.

Fonctionnalités non-implémentées

Malheureusement à ce jour plusieures fonctionnalités ne sont pas implémentées dans notre projet de compilateur. L'affectation multiple suite à un appel de fonction (renvoyant plusieurs variables) n'est effectivement pas possible. Cette commande n'est par exemple pas possible, et renverra un message d'erreur :

- X, Y := (func X), avec func renvoyant 2 variables

Les commandes suivantes sont cependant possibles :

- X := (func X), avec func renvoyant 2 variables (X aura le premier résultat de func)

Enfin, l'affectation multiple est possible, mais nous considérons la chose suivante :

X,Y := A,B équivaut à X := A ; Y := B

Hors ce n'est pas le cas :

X,Y := Y,X n'équivaut pas X := Y ; Y:= X

Rapports d'activité individuelle

Basile Chiquet (responsable de test)

- Conception de la grammaire WHILE en XTEXT avec Emmanuel
- Création de plusieurs fichiers de tests WHILE pour le PrettyPrinter
- Réalisation du pretty printer dans un JAR avec Antoine
- Développement de la classe permettant la génération de code cible JavaScript (JsGenerator) avec Emmanuel
- Documentation du projet avec Emmanuel
- Test de l'interface web

Antoine Gautrain

- Etude d'ANTLR et réalisation du client Node.js appliquant une grammaire simple
- Réalisation d'AbstractGenerator avec les templates Xtend pour produire les premiers fichiers pretty-printés
- Conception de la commande --help pour le pretty printer au format JAR avec Basile
- Développement de l'interface web (design et logique)
- Développement du serveur Java chargé d'écouter sur le port 3333 et de fournir le code JavaScript correspondant au code WHILE transmis
- Aide pour l'aspect JavaScript du projet, objet whileFunctions, métadonnées, etc...

Faustine Godbillot (chef de projet)

- Mise en place des outils
- Répartition et suivi des tâches, rédaction des présentations
- Programmation d'une partie du pretty-printer
- Organisation et création de plusieurs fichiers tests

- Développement des structures de données et traduction en code 3@
- Aide à la création de l'exécutable, à la traduction, et à la création de l'interface web et du serveur

Mame Gueye

- Participation à la rédaction des règles à appliquer pour l'implémentation de la grammaire pour le pretty printer while.
- Suivi de la mise en place du générateur du pretty printer et du code intermédiaire 3 adresses (programmation).
- Participation à la rédaction du document pour le tableau de correspondances while - JavaScript.

Emmanuel Jégo

- Conception de la grammaire de WHILE en XTEXT avec Basile
- Création du pretty-printer avec Antoine
- Début de mise en place d'un second pretty-printer : s'appliquant dans l'éditeur de code avec une combinaison de touches ; abandonné car trop chronophage et hors cahier des charges
- Développement du générateur de code intermédiaire 3@
    - Création des classes nécessaires : tables de fonctions, différentes instructions, …
    - Création du générateur en lui-même
- Développement du générateur de code JavaScript à partir du code 3@ avec Basile
- Participation au développement de la connexion interface web/serveur/compilateur mené par Antoine : partie pretty-printing, partie test du code généré, réalisation de l'animation "? Aide", gestion des erreurs
- Documentation avec Basile

Gwendal Labrue

- Participation à la rédaction des règles à appliquer pour l'implémentation de la grammaire pour le pretty printer while.
- Conception de fonctions-test pour le pretty printer.

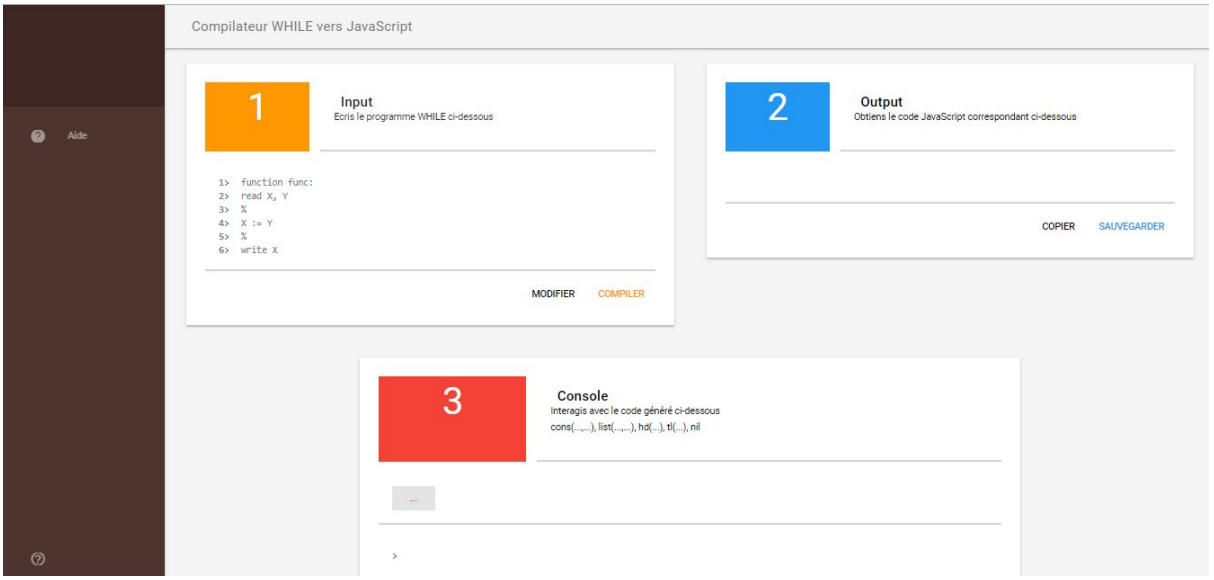
Nicolas Rochat

- Participation à la résolution du problème d'ambiguïté dans la grammaire avec Emmanuel
- Participation à la conception de la génération du code 3@ avec Emmanuel
- Participation à la résolution du problème du code à plat avec Emmanuel

Annexes

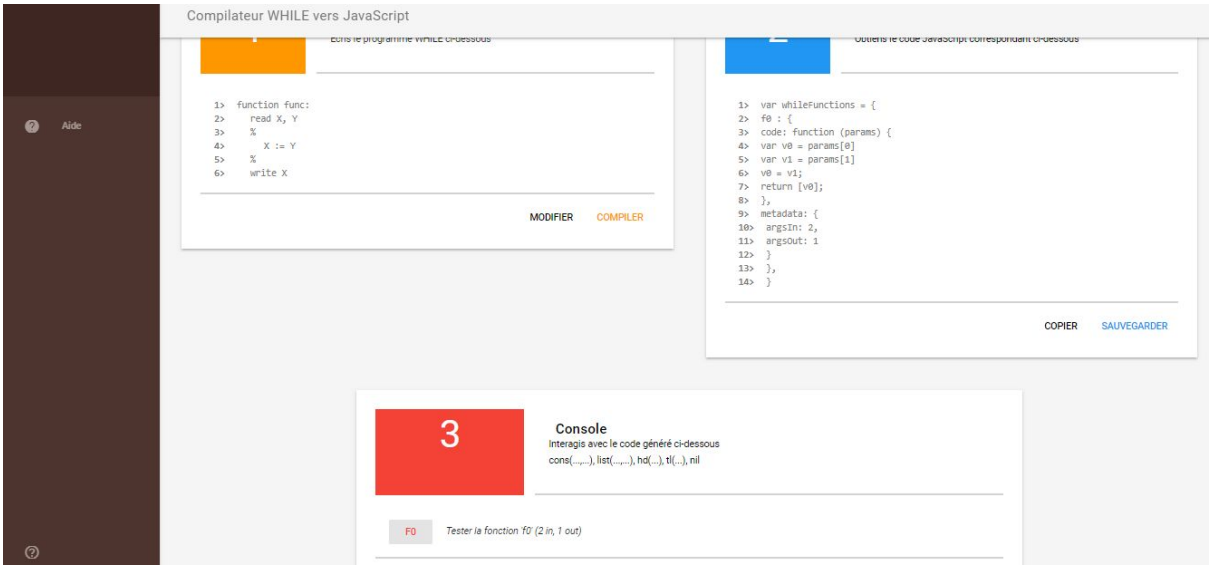
Interface web avant compilation 12
Interface après compilation 13

Interface web avant compilation



Code “Input” non pretty-printé

Interface après compilation



Code “Input” pretty-printé
Code “Output” JavaScript affiché
Fonction “f0” peut être appelée