Compilation
Fiche de Révisions:
Mini-JS-Machine

5 avril 2021

Attention : La compilation décrite ici est compatible avec JavaScript au sens où lorsque le résultat compilé et le JavaScript initial s'éxécutent tous deux sans erreur, alors les deux exécutions doivent être équivalentes. Mais on peut se permetre de rendre plus (fonctionalités non implémentées) ou moins d'erreurs (verifications non implémentées).

1 La Mini-JS-machine minimal

1.1 La sémantique

1.1.1 Types de données.

Dans une case mémoire de la pile on peut trouver plusieurs types de structures, mais on ne peut pas vérifier qu'il s'agit du type attendu, on fait entièrement confiance au programmeur.

Voici les types :

Une instruction. Noté `#i`. Il s'agit d'une des instructions décrites au chapitre idoine et qui déclenche une action spécifique de la machine.

Une valeur. Noté `#v`.C'est une énumération des types suivants, on peut récupérer son type grace à la fonction `x.type` :

- **undifined.** Noté `#u` (un seul élément dedans).
- **doubles.** Noté `#n`.
- **booleen.** Noté `#b`.

Une pointeur. On peut avoir un pointeur sur l'une de ces structures, indiqué `*i`, ou `*v`.

Un nom. Noté `#d`. Il s'agit d'uns string permetant de récupérer une valeur dans un contexte.

Un contexte. Noté `*c`. Un contexte `*c cont` est un dictionnaire (aussi appelé Map, ou table d'association selon les implémentations) qui, à des noms associe des valeurs `#v cont.get(#d var)`.

1.1.2 Les zones mémoires.

La bande de code. Le `CODE` est une bande de calcul en lecture seule. Il s'agit du programme entré traduit en une suite d'instructions (voire la section suivante pour le détails des instructions). Ne contient que des instructions `#i`.

La pile. La `STACK` est une bande de calcul de taille virtuellement infinie et manipulée comme une pile (lifo). Chaque case contient une valeur `#v`.

Les contextes. Les contextes sont stockés dans une zone appelée `CONT`. On les abstrait comme des dictionnaires. [1]

Le reste de la mémoire. Cette machine est abstraite, on peut stocker des structures à des adresses diverses (généralement sur le tas). On appelle `RAM` le reste de la mémoire. [2]

Les pointeurs. Il y a plusieurs pointeurs (adresses) spéciaux que l'on manipule en permanance :

- `SP:*d` pour *Stack Pointer* : pointe sur une adresse de `STACK` correspondant au sommet de pile.
- `PC:*i` pour *Code Pointer* : pointe sur une instruction `CODE[PC]:#i` dans `CODE` correspondant a l'instruction réalisée.
- `CC:*c` pour *Curent Context* : pointe sur le contexte courant `*c`. dans `CONT`.

1.1.3 Initialisation.

Au départ :

- la pile n'a qu'un seul élément qui est `#u Undefinit`, pointée par `SP`,
- le code contient le fichier donné en parametre parsé,
- `PC` pointe sur la première ligne du fichier,
- la `RAM` ne contient que quelques éléments utiles pour les fragments suivants.

1.1.4 Exécution.

La machine lit une instruction sur `CODE[PC]` et l'exécute, ce qui peut changer `SP`, `PC`, `CC`, `STACK`, `CONT` et `RAM`, mais pas `CODE`. Une fois l'instruction effectuée, elle continue jusquà ce que `PC` arrive à la dernière instruction de `CODE` (instruction `HALT`).

1. mais dans l'implémentation de référence, la situation est un peu plus complèxe.
2. même si techniquement tous les autres sont aussi dans la ram...

Si à n'importe quel moment `PC` ou `SP` sort de sa zone allouée, le comportement n'est plus défini ; mais ca ne devrait pas pouvoir arriver.

1.2 Les instructions

1.2.1 Syntaxe pseudo-code

Manipulation de la pile On utilise les fonctions suivantes pour exprimer des manipulations de la pile :

- `Push(#v x):= (STACK[++SP]:=x)` pour pousser une valeur sur la pile.
- `#v Pop := return STACK[SP--]` pour récupérer la tête de la pile et décrémenter cette dernière.
- `#v Pull := return STACK[SP]` pour récupérer la tête de la pile sans décrémenter cette dernière.

Manipulation du contexte Le contexte est un dictionnaire (aussi dit Map, ou table d'association), avec les opérations suivantes :

- `#v Get(#d var)` récupère la valeur associée à `var` dans le contexte. Le comportement n'est pas défini si le nom n'est pas trouvé.
- `Set(#d var,#v val)` modifie la valeur associée à `var` dans le contexte. Si le nom n'est pas trouvé, on force l'association, au nom `var`, de la valeur `val`.

Ces opérations s'applique au contexte courant.

Opérations On suppose toutes les opérations habituelles d'arithmétique ou de comparaison sur les entiers et les flottants, ainsi que l'arithmétique booléenne et le cast des entiers vers les flottants. Attention, pour diférencier les opérations sur les entiers et sur les flottants, on utilise un indice (ex : $+_i$ est la somme d'entier).

Sens d'évaluation Le psedo-code se lit de la droite vers la gauche, par exemple, si j'écrit `Push (Pop -_i Pop)`, celà correspond au Pop du membre droit, puis celui de membre gauche puis la négation puis le Push. Ne vous inquietez pas, il s'agit de l'ordre naturel puisque l'on travail sur une pile, dans l'exemple ci-dessus, on a bien empilé le membre de gauche avant le membre de droit.

1.2.2 Arithmétique

Instruction	sémantique	pile avant	pile après
AddiNb	Push(Pop $+_f$ Pop);	#n:#n:pile	#n:pile
SubsNb	Push(Pop $-_f$ Pop);	#n:#n:pile	#n:pile
MultNb	Push(Pop $*_f$ Pop);	#n:#n:pile	#n:pile
DiviNb	Push(Pop$/_f$Pop);	#n:#n:pile	#n:pile
NegaNb	Push(Pop $*_f$ (-1));	#n:pile	#n:pile
Not	Push(not(Pop));	#b:pile	#b:pile
BoToNb	BoolToDouble(Pop);	#b:pile	#n:pile
NbToBe	DoubleToBool(Pop);	#n:pile	#b:pile

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :

```
PC := PC+1
```

Notez que `Floor` projette un flottant sur le plus grand entier inférieur.

Exemple Encodage (possible) de `3+5*7+2` :[3]

```
CstNb 3          AddiNb
CstNb 5          CstNb 2
CstNb 7          AddiNb
MultNb
```

1.2.3 Comparaison

Instruction	sémantique	pile avant	pile après
Equals	Push(Pop == Pop);	#v:#v:pile	#b:pile
NotEql	Push(Pop $\neq_f$ Pop);	#v:#v:pile	#b:pile
LoEqNb	Push(Pop $\leq_f$ Pop);	#n:#n:pile	#b:pile
GrEqNb	Push(Pop $\geq_f$ Pop);	#n:#n:pile	#b:pile
LoStNb	Push(Pop $<_f$ Pop);	#n:#n:pile	#b:pile
GrStNb	Push(Pop $>_f$ Pop);	#n:#n:pile	#b:pile

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :
`PC := PC+1`

1.2.4 Jumps

Instruction	sémantique	pile avant	pile après
Jump offset	PC := PC + off + 1;	pile	pile

ConJmp offset	if Pop
	then PC := PC + 1;
	else PC := PC + off + 1;
		#b:pile	pile

Case	PC := PC + Floor(Pop) + 1;	#n:pile	pile
Case p	PC := PC + p $*$ Floor(Pop) + 1;	#n:pile	pile

3. CstNb est une instruction que l'on voit plus tard, elle met une constante sur la pile.

Ici, les offsets sont des entiers écrits directement dans le code assembleur. Celà correspond à un entier positif ou négatif indiquant une position dans le code relative à PC. [4]

Le case n'est pas nécessaire au début et peut être remplacé par des ConJump dans la suite, c'est un exemple de commande qui est pratique pour des optimisations ; en particulier pour le typage dynamique il utiliser des Ifs serait beaucoup plus long.

Explication du Jump et du CondJump Jump en ConJump servent à encoder les structures de contrôle : leif, le while et le for principalement.

Le jump est un goto : il déplace la tête de lecture de la machine d'autant qu'indiqué

Le jumpconditionel ConJmp est un "if not goto" : il prend un booléen sur la pile, si c'est un false il se comporte comme un jump et si c'est un true, la machine passe juste à l'instruction suivante.

Exemple Encodage (possible) de `TantQue (3 > 2) {42;}` :

```
CstNb 3      |  ConJmp 3
CstNb 2      |  CstNb 42
GreStR       |  Jump -6
```

1.2.5 Gestion de la pile

Instruction	sémantique	pile avant	pile après
CsteNb x	Push(x);	pile	#n:pile
CsteBo x	Push(x);	pile	#b:pile
CsteUn	Push(Undefined);	pile	#u:pile
Copy	Push(Peak);	X:pile	X:X:pile

Swap	#vx := Pop;
	#vy := Pop;
	Push(x);
	Push(y);
		X:Y:pile	Y:X:pile

Drop	Pop;	X:pile	pile
Noop	ne fait rien	pile	pile
TypeOf	Peak une valeur, rend 0 sur un booleen, 1 sur un flottant, 2 sur une string, 3 sur undefined, 4 sur une cloture, 5 sur un objet,	X:pile	#n:X:pile

Toutes ces opérations sont suivies d'une incrémentation du compteur de code : `PC := PC+1`

Ici, `(X)` et `(Y)` désignent des types quelconques (polymorphisme).

Le `x` de **CstE x** est un entier donné directement dans le code assembleur. On ne met pas de restriction sur sa taille.

4. Dans la pluspart des assembleurs, on utilise des labels qui sont substitués à la volé comme des offsets.

1.2.6 Gestion des variables

Instruction	sémantique	pile avant	pile après
SetVar n	`Set(n,Pop);`	`#v:pile`	`pile`
GetVar n	`Push(Get(n))`	`pile`	`#v:pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code : `PC := PC+1`

Les `n` sont des chaînes de caractères données directement dans le code assembleur et faisant référence à des noms de variables, méthodes, ou classe. On ne met pas de restriction sur leurs tailles mais elles ne peuvent pas contenir d'espace.

1.2.7 Securité et appels système

Instruction	sémantique	pile avant	pile après
Halt	arrête la machine	`pile`	
Error s	renvoit l'erreur s	`pile`	
Assert s	renvoit l'erreur s si Pop est faux	`#b:pile`	`pile`
Print	affiche `Pop` sur le terminal	`#v:pile`	`pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code (lorsque la machine n'est pas arrêtée) :
`PC := PC+1`

Seuls **Print** et **Halt** (fin du programme) sont nécessaires pour le projet. Les autres sont importants pour le débugage.

1.3 Exemple de compilation

Vous êtes libre de compiler un programme comme vous voulez tant que sa sémantique (c'est à dire les couples entrées/sorties) est le même. Néanmoins, pour plus de clarté, nous fournissons ici un exemple de compilation de petit programme.

`Si (x) y = 3 Sinon toto = z ;` peut se compiler en :

```
GetVar x        |   Jump 3
ConJmp 4        |   Drop
Drop            |   GetVar z
CstNb 3         |   SetVar toto
SetVar y        |   Halt
```

2 La Mini-JS-machine avec fonctions

Tout ce qui est donné dans le package 1 reste vrai pour le package 2, sauf si précisé autrement (principalement les contextes).

2.1 La sémantique

2.1.1 Types de données.

Les pile de noms. on considère aussi des piles de noms, celà permet de gérer les arguments des fonctions.

Une cloture. Noté `#l`. C'est un type de donnée `#l x` fonctionnel, défini par un contexte `#c x.cont`, le code d'une fonction `#i x.code` ainsi qu'une pile avec des noms d'arguments. Remarquez que l'on a alors un contexte référencés dans une valeur, on ne travail donc plus avec un seul contexte.

Une continuation. Noté `#t`. C'est une cloture sans argument, mais avec un flag `#b x.err` indiquant si c'est un retour de fonction, un catch.

Les valeurs. Une valeur peut maintenant être une cloture `#l` ou une string, notée `#s`.

Partage de contextes. Deux contextes diférents peuvent partager une partie de leurs variables. Celà veut dire que si on modifie une variable commune dans un contexte, elle sera modifiée dans tous les contextes. On parle de "prototype commun".

2.1.2 Le zones mémoires.

L'arboressence des contextes. Le contexte courant n'est plus le seul contexte. Tous les contextes sont stoqués dans la zone `CONT`.

2.1.3 Les zones de codes.

L'entête du programme, ainsi que les entêtes de classe et de méthodes sont spécifiques : elles définissent des variables, méthodes et classes globales pour le/la programme/classe/méthode.

2.1.4 Les zones de codes.

L'entête du programme, ainsi que les entêtes de classe et de méthodes sont spécifiques : elles définissent des variables, méthodes et classes globales pour le/la programme/classe/méthode.

Entête du programme L'entête du programme total contient les déclarations de variables globales faites avec `Var`, et les déclarations de fonction globales ; car celles-ci son "hosted".

Entête de fonction L'entête de fonction contient les déclarations de variables et fonctions locales.

Contexte global. Le contexte original est un contexte global, partagé par tous les contextes créés par la suite. Celui-ci est modifié chaque fois qu'une variable n'est pas déclarée.

2.2 Les instructions

2.2.1 Syntaxe psedo-code

Manipulation des contextes Le contexte est un dictionnaire (aussi dit Map, ou table d'association), avec les opérations suivantes :

- `#v Get(#d var)` récupère la valeur associée à `var` dans le contexte. Le comportement n'est pas défini si le nom n'est pas trouvé.
- `Insert(#d var, #v val)` associe au nom `var` la valeur `val` dans le contexte courant, si le nom est déjà référencé et partagé par un autre contexte, la nouvelle valeur cache l'ancienne, mais ne modifie pas l'ancienne valeur, au sens où l'autre contexte pointant sur cette ancienne valeur ne verra aucune modification.
- `Set(#d var, #v val)` modifie la valeur associée à `var` dans le contexte. Si le nom n'est pas trouvé, on force l'association, au nom `var`, de la valeur `val` **dans le contexte global**.
- On utilise `#c CopyCont` pour déclarer une copie du contexte.

Ces opérations s'applique au contexte courant si utilisées comme fonction et non comme métodes. Par exemple `Get("toto")` récupère la valeur de toto dans le contexte courant `CC`, mais `autreContext.Get("toto")` récupère la valeur de toto dans le contexte `*c autreContext`. Ces opérations s'appliquent au contexte courant.

Manipulation des piles de noms Les fonctions `Pop`, `Push` et `Pull` s'appliquent aussi à des piles de noms.

2.2.2 Appels de méthodes

Instruction	sémentique	pile avant	pile après
StartCall	`Pull.setContext(NewContext(CC))`	`#l:pile`	`#l:pile`

SetArg	`#v v = Pop`
	`#l clot = Pull`
	`#n n = clot.args.Pop`
	`clot.cont.Insert(n, v)`
	`PC := PC + 1`
		`#v:#c:pile`	`#c:pile`

Call	`#l clot = Pop`
	`Push(NewContinuation{cont = CC,`
	`code = PC,`
	`err = 0})`
	`CC := clot.cont`
	`PC := clot.code`
		`#l:pile`	`#t:pile`

Return	`#v res = Pop;`
	`do {#t continue = Pop;}`
	`while (continue.err)`
	`CC := continue.cont`
	`PC := continue.code`
	`Push(res)`
		`X:autre@#t:pile`	`X:pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :
`PC := PC+1`

2.2.3 Déclarations de méthodes et d'arguments

Instruction	sémentique	pile avant	pile après
DclVar n	`Insert(n, undefind)`	`pile`	`pile`
NewClo off	`Push(NewCloture{cont = CopyCont,` `code = PC + off + 1})`	`pile`	`#l:pile`
DclArg n	`Pull.args.Push(n)`	`#l:pile`	`#l:pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :
`PC := PC+1`

2.2.4 Exceptions

Instruction	sémentique	pile avant	pile après

Throw	`#v res := Pop`
	`docontinue := Pop;`
	`while(Type(continu) ≠ Continuation`
	`  || not continue.err)`
	`CC := continue.cont`
	`PC := continue.code`
	`Push(res)`
		`X:autre@#t:pile`	`X:pile`

Continue pos b	`Push(NewContinuation{cont = CC,`
	`  code = pos,`
	`  err = b})`
		`pile`	`#t:pile`

Catch off	`Push(NewContinuation{cont = CC,`
	`  code = PC + off + 1,`
	`  err = true})`
		`pile`	`#t:pile`

Finally off	`Push(NewContinuation{cont = CC,`
	`  code = PC + off + 1,`
	`  err = false})`
		`pile`	`#t:pile`

Local off	$CC = NewContext(CC)$	`pile`	`pile`
Global off	$CC = CC.oldContext$	`pile`	`pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :
`PC := PC+1`

2.2.5 Arithmétique des Strings

Instruction	sémantique	pile avant	pile après
Concat	`Push(Pop +s Pop);`	`#s:#s:pile`	`#s:pile`
CstStr x	`Push(x);`	`pile`	`#s:pile`
StToNb	`StringToDouble(Pop);`	`#s:pile`	`#n:pile`
NbToSt	`DoubleToString(Pop);`	`#n:pile`	`#s:pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :
`PC := PC+1`

2.2.6 Comparaison

Instruction	sémantique	pile avant	pile après
LowEqS	`Push(Pop ≤s Pop);`	`#s:#s:pile`	`#b:pile`
GreEqS	`Push(Pop ≥s Pop);`	`#s:#s:pile`	`#b:pile`
LowStS	`Push(Pop <s Pop);`	`#s:#s:pile`	`#b:pile`
GreStS	`Push(Pop >s Pop);`	`#s:#s:pile`	`#b:pile`
IsEmpt	`Push(Pop ≠s "");`	`#s:pile`	`#b:pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :
`PC := PC+1`

3 La Mini-SJ-machine avec objets

3.1 La sémantique

3.1.1 Types de données.

Un objet/prototype. Noté `#o`. Un objet `x : o` est fondamentalement la même chose qu'un contexte, c'est à dire que c'est un dictionnaire associant des noms à des valeurs. Il peut avoir une implémentation diférentes pour des raisons d'efficacité. L'objet null est l'objet vide.

3.2 Les instructions

3.2.1 Syntaxe psedo-code

On utilise `NewObjet{}` pour déclarer un nouvel objet vide.

3.2.2 Objets

Instruction	sémentique	pile avant	pile après
Point nom	`Push(Pop.Get(nom))`	`#o:pile`	`#v:pile`
Null	`Push(Null)`	`pile`	`#o:pile`
NewObj	`Push(NewObjet{})`	`pile`	`#o:pile`
SetObj nom	`Pull.cont.Set(nom, Pop)`	`#v:#o:pile`	`#o:pile`

SetIn nom	`#v v = Pop`
	`#l clot = Pull`
	`clot.cont.Insert(nom, v)`
		`#v:#c:pile`	`#c:pile`

Mise à jour : En JS, null et l'objet vide sont différent (tous deux différents de undefined !!), on a donc une instruction `Null` et une instruction pour `NewObj`

4 La Mini-JS-machine avec classes

4.1 Différences avec JS

Vous aurrez peut-être remarqué qu'en javascript les fonctions sont des objets et ne sont pas des types promitifs (en JS, les seuls types primitifs sont *number, string* et *object*). Nous les avons utiliser comme primitifs afin de pouvoir les inclure dans les package avant les objets.

Il s'agit d'une décision partielement arbitraire mais motivé par la remarque que *in fine* il faut implémenter les cloture dans la machine, et qu'elles sont plus aisées à implémenter comme type primitifs. Il faut aussi comprendre que les classes sont arivées très tard en JS (en 2017 avec ES6), avant on utilisait les fonctions à la place des classes (en considérant qu'une classe était entièrement définie par son constructeur), ce qui rendait très naturel le fait que ce soit lui même un objet.

La contre-partie est qu'en mini-JS, il n'y a pas moyen d'utiliser une fonction comme s'il s'agissait d'une classe. Celà rends aussi impossible la prospection interne des fonctions (récupération du nom de la fonction, utilisation de l'environement d'appel...). [5]

Autre différences : nous ne demandons pas l'implémentation de toutes les méthodes prédéfinies de la classe objet... Ici, le prototype d'un nouvel objet ou d'une nouvelle classe sans héritage est l'objet Null, alors que normalement ce doit être le prototype par défaut dont le prototype est lui-même...

4.1.1 Types de données.

Classes Une classes `maClasse` sont des objets avec un attribut `nom : "maClasse"` et un autre attribut (mal)nomé `prototype`. Ce dernier définit l'objet qui constituera le prototype `__proto__` de l'objet créé par le constructeur `new maClasse()`.

4.2 Les instructions

4.2.1 Objets

Instruction	sémentique	pile avant	pile après
Protot	`Push(NewObjet{__proto__ = pop()})`	`#o:pile`	`#o:pile`
GetPrt	`Push(pop().__proto__)`	`#o:pile`	`#o:pile`

5. Ce qui est, à mon sense, une bonne chose, car le nom avec lequelle la fonction a été créé doit rester privé, à moins de faire du débuggage

5 Instructions suplémentaires pour les optimisations

Instruction	sémentique	pile avant	pile après

TlCall	`#l clot = Pop`
	`CC := clot.cont`
	`PC := clot.code`
		`#l:pile`	`pile`

RetOpt `#v res = Pop;`	`X:autre@#t:pile`
	`cont`
	`CC := continue.cont`
	`PC := continue.code`
	`Push(res)`
		`X:pile`
		`cont`
			`#t continue = Pop;`

RetZer	`do {#t continue = Pop;}`
	`cont`
	`CC := continue.cont`
	`PC := continue.code`
		`autre@#t:pile`
		`cont`
			`pile`
			`while (not continue.flag)`

RetOpZ	`#t continue = Pop;`
	`cont`
	`PC := continue.code`
	`Push(res)`
		`autre@#t:pile`
		`cont`
			`pile`
			`CC := continue.cont`

SetVaD n	`Set(n, Pull);`	`#v:pile`	`pile`
SetVarStack i **GetVarStack i**	`stack[sp − i] := Pull;` `Push(stack[sp − i]);`	`#v:pile` `#v:pile`	`pile` `pile`