Projet de compilation
Compilateur de Javascript
(vers un assembleur abstrait)

29 avril 2020

Attention : Ce document est amené à changer en fonction de l'évolution du cours, en particulier le barème et les features obligatoires peuvent être modifiées (mais uniquement à votre avantage).

Le projet consiste à compiler une version francisée du Javascript dans un assembleur abstrait maison (voir du web-assembly pour les plus motivés).

Le langage et les outils pour écrire le compilateur sont libres. Mais si vous utilisez d'autres méthodologies et d'autres outils que ceux vu en cours, il vous faudra les argumenter proprement.

Il y a 5 *Fragments* de javascript possibles, le minimal (0-ième) noté sur 10, le premier noté sur 14, le second sur 18, le troisième sur 12,5 (ou son alternatif, le 3bis), les troisième+son alternative (3+3bis) sur 22, et le dernier sur 23,5. Chaque *Fragment* correspond à un fragment du langage JS. Pour chaque *Fragment*, il y a 3 étapes, correspondant aux étapes du schéma de compilation vu en cours, vous n'êtes pas obliger d'aller jusqu'au bout, mais faites-les dans l'ordre.

Vous pouvez rendre jusque 3 projets par binomes : un par étape. Un *Fragment* inférieur ne sera considéré que sur les étapes non implémentés dans les *Fragments* supérieur. Exemple : si un binome rend 3 projets, le *Fragment* 1 jusqu'à l'étape 3 le *Fragment* 2 jusque l'étape 2 et le *Fragment* 3 jusque l'étape 3, alors l'étape 1 sera noté sur le *Fragment* 3, l'étape 2 sur le *Fragment* 2 et l'étape 3 sur le *Fragment* 1.

Voici les points attribuables par phase et par *Fragment* :

	Lexeur	Parseur	code abstraite	total
Fragment 0	2.5	3.5	4	10
Fragment 1	3	5.5	5.5	14
Fragment 2	3.5	7	7.5	18
Fragment 3 ou 3bis	4	8	8.5	20.5
Fragment 3 + 3bis	4	8.5	9.5	22
Fragment 4	4	8.5	11	23.5

Attention : au rendu, veuillez bien indiquer quel projet correspond à quel *Fragment* et quelle étape.

1 Description rapide des *Fragments*

1.1 *Fragment* 0

Dans un premier temps, on considérera un langage minimal, sans aucune des "features" de *Javascript* :

- Contrairement à JS, on utilisera la casse pour séparer les mot-clefs des variables : les mot-clefs commencent par une majuscule et les variables par une minuscule.
- Deux types de base : `Nombre` (correspond à `number`) et `Bool`. En javascript, il n'y a pas d'entiers, que des flottants. Pour l'instant on ne demande que les écritures entières (ex : `34`, `-12`, `0045`) et à virgule fixe (ex : `3.4`, `-1.2`, `0.045`).
- Pas de conversion de type implicite.
- On peut modifier une variable à l'aide du "`x = **`". (bonus) Vous pouvez ajouter les initialisateurs avec opérations *=, +=...
- On peut déclarer une variable en l'instanciant simplement à l'aide de `x = **`. Attention, il ne s'agit pas du `var`, les variables ne sont pas "hoiseted", c'est à dire que les variables doivent être déclarées avant d'êtres utilisées ; de plus, elles sont toujours globales.
- On a quelques opérations arithmétiques sur les nombres et les booléens. Attention toutes fois aux priorités ! (bonus) De base on ne demande pas toutes les opérations (en particulier pas le ==, juste le ===), vous pouvez en rajouter...

1.2 *Fragment* 1

Dans un premier temps, on considérera un langage minimal, sans aucune des "features" de *Javascript* :

- On demande le `Si(..){..}Sinon{..}` et optionnellement le `Si(..){..}`. Attention aux priorités.
- Ajoutez le `TantQue(..){..}` (le `while`), le `Faire{..}TantQue(..)` (le `do while`) et le `Pour(..,..,..){..}` (le `for` dans le cas de base).
- Une fonction `ecrire(..)`, qui est un appel système et qui lit n'importe quel type.
- On veut pouvoir écrire des commentaires uniligne (de la forme `// mon commentaire`) et multiligne (de la forme `/* mon commentaire */`.

1.3 *Fragment* 2

On ajoute ici quelques features identitaires de javascript :

- On ajoute un type de base : les `String`, mais sans les caractères échappés,
- Une variable peut être déclarée à l'aide du `Var` sans être instancée imédiatement. On peut maintenant déclarer les variables avec `Var` n'importe où, elle sera remontée au global ou en tête de la fonction courante.

— On a un autre type suplémentaire : le type `Indef` (correspond à `undefined`) pour les variables déclarées non instanciées.
— On ajoute le `+` sur les strings
— On demande maintenant de faire de la conversion implicite (testez bien tous les cas !!).
— On permet l'écriture de fonctions, introduites avec le mot clé `Fonction` et avec le résultat retourné par `Retourne`. Attention, en JS, une fonction peut être appellée avec un nombre d'arguments différent de celui définit...

1.4 *Fragment* 3

Finalement on ajoute un peut de fonctionnelle et d'objet :
— On doit pouvoir utiliser les charactères d'échapements dans les strings.
— (bonus) on peut vouloir ajouter les template strings
— Il faut ajouter les lambda expressions et les clôtures : les fonctions sont des valeurs dont le type est `Fonc`.
— On vérifiera que l'on supporte bien la récursivité.
— On va aussi ajouter les objets, mais pas les classes, c'est à dire que l'on utilise juste `{attr : **; meth : **}`. Par contre, on n'utilise pas les objets de javascript. Les changements sont les suivants :
  — on ne peut pas avoir de parametre/méthode d'un objet qui ai un nom réservé ! En JS c'est possible, ils font ca car un objet peut être récupérer depuis du html ou du JSON dans lesquels ces mots clés ne sont pas réservés. (bonus) Implémentez la version de JC [1]
  — le `this` (que l'on appelera `this`) est juste une variable accessible depuis les méthodes d'un objet et désignant celui-ci (comme en Java). En JC, le sense de `this` change lorsque l'on rentre dans une fonction ou dans une *lambda*-expression (et de manière différente !!), ils auraient due utiliser 3 mots clés différents...
  — On demande aussi le `Switch`.

  N'oubliez pas d'implémenter l'objet `Nul`.
— Pour le fragment 3, on doit pouvoir utiliser des noms de variable avec majuscule.

1.5 *Fragment* 3bis

Il s'agit de rajouter les exceptions. Auparavant, cette feature était en fragment 2, mais avec le cours à distance, elle n'est pas ficile a comprendre et donc à été reculée.

1.6 *Fragment* 4

On va maintenant ajouter les classes et les prototypes :
— On rajoute les classes, avec leurs contructeurs, méthodes, getteurs, setteurs...

1. Utilisez des token spécifiques pour ".atribut" et "atribut :"

— Une classe peut avoir une méthode appellée `Constructeur` qui est exécuté à la création d'objet ; sans constructeur, on instancie juste les atributs et méthodes par défaut.
— On a accès au prototype d'un objet.
— Une fonction est maintenant une classe créée à partir d'un prototype prédéfinit.
— (bonus) Implémentez le `this` de JC [2]
— (bonus) On poura ajouterer la syntaxe des tableaux. Les tableaux étant des objets, on peut écrire une bibliothèque par défaut. On demande la syntaxe `[1,2,3]` pour la création, et l'utilisation de la boucle `Pour (.. Dans ..) {..}`

1.7 Bonus

Les Parties grisés font partie de la syntaxe, mais ne sont pas demandés, les ajouter vous voudra un petit bonus.

Après avoir implémenter tout ça, vous êtes libre de faire plus. Cela sera ajouter à votre note, mais comptera moins à difficulté égale. En particulier, vous pouvez introduire des messages d'erreur ou des optimisations à votre code.

Dans tous les cas, ces bonus ne sont pas dans le barème et sont à la discrétion du correcteur.

2 Les étapes à implémenter

2.1 Lexeur

Il s'agit, ici, de produire (ou d'écrire) un lexeur pour les tockens de la restriction de langage considéré.

En plus des mots clefs, des opérations et des fiférentes parenthèses, on aura :
— *Fragment* 0 :
les <NOMBRE> reconnus par `[0-9]+'.'[0-9]*`,
les <IDENT> reconnues [3] par `[a-z][a-z,A-Z,0-9,_,'-']`,

— *Fragment* 1 :
les <BOOLEEN> reconnus par `["Vrais","Faux"]`,
les commentaires unilignes et multilignes.
— *Fragment* 2 :
les <STRING> reconnus par `'"'[' ',!,#-z]*'"'`,
les <NOMBRES> doivent se raprocher au maximum de ceux de JC (à vous de tester),
— *Fragment* 3 :
les <STRING> doivent se raprocher au maximum de celles de JC (à vous de tester).

2. Modifiez l'environement lors de l'appel de fonction (hors méthodes...)
3. <IDENT> pour "identifiant", reconnait tout ce qui est nom de variable, fonction, classe... le mot "identifiant" est celui utilisé dans la litérature.

2.2 Parseur

Il s'agit, ici, de produire (ou d'écrire) un parseur pour les tockens de la restriction de langage considéré.

On va préciser la grammaire de nos différentes restrictions. [4]

Code couleur : En gris, vous avez les features optionnelles ; en rouge, vous avez les nouvelles features obligatoires du *Fragment* ; et en rose, vous avez les nouvelles features optionnelles du *Fragment*.

Attention : N'oubliez pas de bien définir vos règles de priorité/associativité pour vos opérateurs...

2.2.1 Grammaire du *Fragment* 0

<programme> ::= <commande> | <commande> <programme>

<commande> ::= ;
| <expression> ;
| <IDENT> = <expression> ;

<expression> ::= <NOMBRE> | <IDENT>
| (<expression>)
| <expression> + <expression>
| <expression> - <expression>
| <expression> * <expression>
| <expression> / <expression>
| <expression> ===<expression>
| <expression> ? <expression> : <expression>

4. Celles-ci sont un peu différentes de la grammaire officielle du langage que l'on peut trouver ici, mais essenciellement similaires, on utilise simplement une restrictions (et des simplifications à droite à gauche).

2.2.2 Grammaire du *Fragment* 1

<programme> ::= <commande> | <commande> <programme>

<commande> ::= `;`
| `{` <programme> `}`
| <expression> `;`
| `Si` (<expression>) <commande>
| `Si` (<expression>) <commande> `Sinon` <commande>
| `TantQue (` <expression> `)` <commande>
| `Faire` <commande> `TantQue (` <expression> `)`
| `Pour (` <expression> `;` <expression> `;` <expression> `)` <commande>
| `ecrire(` <expression> `) ;`

<expression> ::= <NOMBRE> | <BOOLEEN> | <IDENT>
| `(`<expression>`)`
| <op_unair_L> <expression>
| <expression> <op_binair> <expression>
| <expression> `?` <expression> `:` <expression>
| <IDENT> <assigne> <expression>

<op_unair> ::= `-` | `!` | `Typeof`

<op_binair> ::= `+` | `-` | `*` | `/` | `%` | `===` | `>` | `!==` | `&&` | `||` | `**` | `>=` | `<=`

<assigne> ::= `=` | `+=` | `*=` | `-=` | `/=` | `%=` | `**=`

2.2.3 Grammaire du *Fragment* 2

```
<programme>     ::=  <commande> | <commande> <programme>

<commande>      ::=  ;
                   | { <programme> }
                   | <affect_expr> ;
                   | Fonction <IDENT> (<arguments>) { <programme> }
                   | Si (<expression>) <commande>
                   | Si (<expression>) <commande> Sinon <commande>
                   | Break
                   | TantQue ( <expression> ) <commande>
                   | Faire <commande> TantQue ( <expression> )
                   | Retourner ( <expression> ) ;
                   | Pour ( <affect_expr> ; <expression> ; <expression> ) <commande>

<affect_expr>   ::=  <expression>
                   | Var <IDENT> = <expression>  | Var <IDENT>

<expression>    ::=  <NOMBRE> | <BOOLEEN> | <STRING> | <IDENT>
                   | (<expression>)
                   | <IDENT>(<expressions>)
                   | <op_unair> <expression>
                   | <expression> <op_binair> <expression>
                   | <expression> ? <expression> : <expression>
                   | <IDENT> <assigne> <expression>

<expressions>   ::=  ϵ | <expression> , <expressions>

<op_unair>      ::=  - | ! | Typeof

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

<assigne>       ::=  = | += | *= | -= | /= | %= | **=

<arguments>     ::=  ϵ | <IDENT> | <IDENT> , <arguments>
                   | <IDENT>=<expression> | <IDENT>=<expression> , <arguments>
```

2.2.4 Grammaire du *Fragment* 3

<programme> ::= <commande> | <commande> <programme>

<commande> ::= ; | { <programme> } | <affect_expr> ;
| Fonction <IDENT> (<arguments>) { <programme> }
| Si (<expression>) <commande> | Si (<expression>) <commande> Sinon <commande>
| Switch (<expression>) { <cas_switch> }
| TantQue (<expression>) <commande>
| Faire <commande> TantQue (<expression>)
| Retourner (<expression>) ;
| Pour (<affect_expr> ; <expression> ; <expression>) <commande>

<affect_expr> ::= <expression>
| Var <IDENT> = <expression> | Var <IDENT>

<cont_objet> ::= <IDENT> : <expressions> | <IDENT> : <expressions> , <cont_objet>

<expression> ::= <NOMBRE> | <BOOLEEN> | <STRING> | <IDENT>
| (<expression>)
| <expressions> (<expressions>)
| <expression>.<IDENT>
| {<cont_objet>} | Nul
| <op_unair> <expression> | <expression> <op_binair> <expression>
| <expression> ? <expression> : <expression>
| <IDENT> <assigne> <expression>
| (<arguments>) => <expression> | <argument> => <expression> ;
| (<arguments>) => {<programme>} | <argument> => {<programme>} ;

<cas_switch> ::= ϵ | Cas <expression> : <programme> <cas_switch> | Cas Defaut : <programme> <cas_switch>

<expressions> ::= ϵ | <expression> , <expressions>

<op_unair> ::= - | ! | Typeof

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

<assigne> ::= = | += | *= | -= | /= | %= | **=

<arguments> ::= ϵ | <IDENT> | <IDENT> , <arguments>
| <IDENT>=<expression> | <IDENT>=<expression> , <arguments>

2.2.5 Grammaire du *Fragment* 3bis

<programme> ::= <commande> | <commande> <programme>

<commande> ::= `;` | `{` <programme> `}` | <affect_expr> `;`
| `Fonction` <IDENT> `(`<arguments>`)` `{` <programme> `}`
| `Si` `(`<expression>`)` <commande> | `Si` `(`<expression>`)` <commande> `Sinon` <commande>
| `Switch` `(` <expression> `)` `{` <cas_switch> `}`
| `Essayer` `{`<commande>`}` `Rattraper` `(`<IDENT>`)` `{`<commande>`}`
| `Essayer` `{`<commande>`}` `Rattraper` `(`<IDENT>`)` `{`<commande>`}` `Finalement` `{`<commande>`}`
| `Essayer` `{`<commande>`}` `Finalement` `{`<commande>`}`
| `Lancer` <expression> `;`
| `TantQue` `(` <expression> `)` <commande>
| `Faire` <commande> `TantQue` `(` <expression> `)`
| `Retourner` `(` <expression> `)` `;`
| `Pour` `(` <affect_expr> `;` <expression> `;` <expression> `)` <commande>

<affect_expr> ::= <expression>
| `Var` <IDENT> `=` <expression> | `Var` <IDENT>

<cont_objet> ::= <IDENT> `:` <expressions> | <IDENT> `:` <expressions> `,` <cont_objet>

<expression> ::= <NOMBRE> | <BOOLEEN> | <STRING> | <IDENT>
| `(`<expression>`)`
| <expressions> `(`<expressions>`)`
| <expression>`.`<IDENT>
| `{`<cont_objet>`}` | `Nul`
| <op_unair> <expression> | <expression> <op_binair> <expression>
| <expression> `?` <expression> `:` <expression>
| <IDENT> <assigne> <expression>
| `(`<arguments>`)` `=>` <expression> | <argument> `=>` <expression> `;`
| `(`<arguments>`)` `=>` `{`<programme>`}` | <argument> `=>` `{`<programme>`}` `;`

<cas_switch> ::= ϵ | `Cas` <expression> `:` <programme> <cas_switch> | `Cas Defaut :` <programme> <cas_switch>

<expressions> ::= ϵ | <expression> `,` <expressions>

<op_unair> ::= `-` | `!` | `Typeof`

<op_binair> ::= `+` | `-` | `*` | `/` | `%` | `===` | `==` | `>` | `!==` | `&&` | `||` | `**` | `>=` | `<=`

<assigne> ::= `=` | `+=` | `*=` | `-=` | `/=` | `%=` | `**=`

<arguments> ::= ϵ | <IDENT> | <IDENT> `,` <arguments>
| <IDENT>`=`<expression> | <IDENT>`=`<expression> `,` <arguments>

2.2.6 Grammaire du *Fragment* 4

<programme> ::= <commande> | <commande> <programme>

<commande> ::= ; | { <programme> } | <affect_expr> ;
| Fonction <IDENT> (<arguments>) { <programme> }
| Class <IDENT> { <methodes> }
| Class <IDENT> Etend <expression> { <methodes> }
| Si (<expression>) <commande> | Si (<expression>) <commande> Sinon <commande>
| Switch (<expression>) { <cas_switch> } | Break
| Essayer {<commande>} Rattraper (<IDENT>) {<commande>}
| Essayer {<commande>} Rattraper (<IDENT>) {<commande>} Finalement {<commande>}
| Lancer <expression>;
| TantQue (<expression>) <commande>
| Faire <commande> TantQue (<expression>)
| Retourner (<expression>) ;
| Pour (<affect_expr> ; <expression> ; <expression>) <commande>

<affect_expr> ::= <expression> | Var <IDENT> = <expression> | Var <IDENT>

<methodes> ::= ϵ | <methode> <methodes>

<methode> ::= <IDENT> | <IDENT>(<arguments>) { <programme> } | <IDENT> = <expression> ;
| Get <IDENT> () { <programme> } | Set <IDENT> (<IDENT>) { <programme> }

<cont_objet> ::= <IDENT> : <expressions> | <IDENT> : <expressions> , <cont_objet>

<expression> ::= <NOMBRE> | <BOOLEEN> | <STRING> | <IDENT>
| (<expression>) | [<expressions>]
| <expressions> (<expressions>) | <expression>.<IDENT>
| {<cont_objet>} | Nul | New <expression>
| <op_unair> <expression> | <expression> <op_binair> <expression>
| <expression> ? <expression> : <expression>
| <IDENT> <assigne> <expression>
| (<arguments>) => <expression> | <argument> => <expression>
| (<arguments>) => {<programme>} | <argument> => {<programme>}

<cas_switch> ::= ϵ | Cas <expression> : <programme> <cas_switch> | Cas Defaut : <programme> <cas_switch>

<expressions> ::= ϵ | <expression> , <expressions>

<op_unair> ::= - | ! | Typeof

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

<assigne> ::= = | += | *= | -= | /= | %= | **=

<arguments> ::= ϵ | <IDENT> | <IDENT> , <arguments>
| <IDENT>=<expression> | <IDENT>=<expression> , <arguments>

2.2.7 Grammaire réduite

Version réduite avec des expressions régulières du fragment considéré (un tout petit peu élargi car on avait de la place). La syntaxe relative aux rexpressions régulière est en bleu de sorte à la distinguer des symboles-tockens en noir.

<programme> ::= <commande>*

<commande> ::= ; | { <programme> } | <affect_expr> ;
| Fonction <IDENT> (<arguments>) { <programme> }
| Class <IDENT> (Etend <expression>)? { <methode>* }
| Si (<expression>) <commande> (Sinon <commande>)?
| Switch (<expression>) { <cas_switch> } | Break
| Essayer {<commande>} (Rattraper (<IDENT>) {<commande>})? (Finalement {<commande>})?
| Lancer <expression> ;
| TantQue (<expression>) <commande>
| Faire <commande> TantQue (<expression>)
| Retourner (<expression>) ;
| Pour (<affect_expr> ; <expression> ; <expression>) <commande>

<affect_expr> ::= <expression> | Var <IDENT> (= <expression>)?

<methode> ::= Statique? <IDENT> (ϵ | (<arguments>) { <programme> } | = <expression> ;)
| Get <IDENT> () { <programme> } | Set <IDENT> (<IDENT>) { <programme> }

<cont_objet> ::= (<IDENT> : <expressions>)+

<expression> ::= <NOMBRE> | <BOOLEEN> | <STRING> | <IDENT>
| (<expression>) | [<expressions>]
| <expressions> (<expressions>) | <expression>.<IDENT>
| {<cont_objet>} | Nul | New <expression>
| <op_unair> <expression> | <expression> <op_binair> <expression>
| <expression> ? <expression> : <expression>
| <IDENT> <assigne> <expression>
| ((<arguments>) | <argument>) => (<expression> | {<programme>})

<cas_switch> ::= (Cas (<expression> | Defaut) : <programme>)*

<expressions> ::= ((<expression> ,)* <expressions>)?

<op_unair> ::= - | ! | Typeof

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

<assigne> ::= = | += | *= | -= | /= | %= | **=

<arguments> ::= ((<IDENT>(=<expression>)?,)* <IDENT>(=<expression>)?,)?

2.3 Interpreteur (Bonus)

Il s'agit d'une phase bonus : écrivez un Interpreteur de JS dans le langage utilisé qui s'exécute à partir de votre arbre syntaxique abstrait.

2.4 Code abstrait

Voir le cours sur les machines abstraites.

3 Les points un peu tricky en détails

3.1 Associativité des opérateurs

Pour touts les opérateurs binaires utilisés dans les grammaires ci-dessus, il n'y a pas de rêgle d'associativité associée, ces dernières sont donc ambigües.

Affin de coriger ça, il faut définir une rêgle d'associativité, en l'occurence tous nos opérateurs ont une associativité gauche, mais je vous encourage à le vérifier (sur les flotants, la somme n'est pas associative, vous pouvez donc tester sur un script JC).

Pour savoir comment encoder l'associativité, se référer au TP 1 (dès le départ pour Java, et en exercice 3 pour les autres).

3.2 Priorité des opérateurs

Vous aurrez sans doute remarqués, l'utilisation du non terminal `<op\_binair>` sans aucune relation de priorité entre les opérations conduit à des résultats étranges. En fait, avec juste l'associativité gauche, $3 + 5 * 2$ est identifié en $(3 + 5) * 2$, les deux opérateurs sont mis dans un même sac.

Pour éviter ça, il faut utiliser la priorité, pour ça, se référer au TP 1 (dès le départ pour Java, et en exercice 3 pour les autres).

Attention, les rêgles depriorité ne s'appliquent pas aux opérateurs, mais aux rêgles qui les construisent ; ainsi la rêgle `<expression>` $\mapsto$ `<expression> <op_binair> <expression>` est une seule rêgle, elle a donc un seul niveau de priorité et toutes les `<op_binair>` seront traités sur un même niveau. Il faut donc faire une raigle binaire pour chaque opération binaire, ou au moins une rêgle binaire pour chaque niveau de priorité...

Pour savoir quel opérateur estprioritaire sur quel opérateur, faites des tests JS !

3.3 Le ifthen et le ifthenelse

Remaquez que le `Si(_){_}` est optionel, contrairement au `Si(_){_}Alors{_}`. C'est parce qu'il y a ici une rêgle de priorité non trivial : le `Si(_){_}Alors{_}` est prioritaire sur le `Si(_){_}`, mais puisque ce n'est pas un opérateur infixe, le trick que l'on utilise dans le TP1 exercice 1 pour traiter les priorité impicitement (avec un nouveau non terminal par niveau de priorité) ne marche plus. Ce n'est

pas très grave pour ceux qui utilisent un générateur LALR (sous C, OCaml ou Haskell) mais ceux sur Java, dont le générateur est LL, ne pouront pas encoder le `Si(_){_}`.

3.4 Assignation comme expression

En JS, l'assignation est une expression ; en miniJS ce n'est pas demandé au fragment 0, mais ça l'est au fragment 1. [5] Celà signifie que l'on peut théoriquement écrire `z = 3*(x = 5)` ce qui assigne 5 à x, puis assigne 15 à z...

Dans la machine, il y avait un bug dans la première version (et je ne pouvais pas l'enlever pour la rétro-compatibilité) : le `SetVar` aurait due laisser la valeur de la variable sur la pile, car c'estle résultat de l'expression en question. Dans certains cas il faudra donc faire un `Copy` avant ou utiliser la commande `SetVar2` qui est une correction de `SetVar` pour respecter toutalement la sémentique.

3.5 Expressions comme commandes

En JS, toute expression est une commande. Dans la pratique, on utilise surtout l'assignement comme commande, mais parfois on peut, par exemple, utiliser une fonction dont on ne s'intéresse pas au résultat.

Celà n'a pas beaucoup d'incidence, le seul soucis, c'est dans le Fragment 3bis. En effet, les exceptions demandent une gestion fine dela pile, etil ne doitpas y avoir de polution. Or si j'écris `21*2;` celà met sur la pile un nombre qui va y rester etla poluer si on ne l'enlève pas. Il faut donc faire un `Drop` derière pour la nétoyer (mais uniquement si on n'a pas utilisé SetVar à la fin...)

5. par contre, ce ne sera quasiment pas punie au frament 1