

This assignment will be closed on June 06, 2023 (23:59:59).

On June 06, 2023 (13:39:52), we had submissions for the following questions: 1, 2, 3, 4 — [Details](#) ➔  
(/agns/INF371/TD06/2022/uploads/:my/).

# Compilation de mini-while vers la XVM

*Pierre-Yves Strub (<http://strub.nu/>).*

- 
- [Installation de la bibliothèque de support](#)
  - [Implémentation de référence](#)
  - [Concernant les tests automatisés](#)
  - [Production d'un flux d'opcode](#)
  - [Compilation des expressions](#)
  - [Compilation des instructions](#)
  - [Procédures](#)
  - [Enregistrements extensibles](#)
    - [Enregistrements en While](#)
    - [Représentation mémoire](#)
    - [Intégration du nouveau typeur à votre projet](#)
    - [Travail demandé](#)
  - [Optimisation des appels terminaux](#)
- 

On se propose d'écrire dans ce TD un mini-compilateur pour le langage `While`, un langage de programmation procédural. La machine cible est la `XVM` qui a été présentée en cours et dont vous trouverez une description détaillée dans le polycopié du cours. Une implémentation en ligne de la `XVM` est fournie à l'adresse suivante:

<http://www.enseignement.polytechnique.fr/informatique/INF371/xvm/>  
(<http://www.enseignement.polytechnique.fr/informatique/INF371/xvm/>).

Afin de tester votre compilateur, on vous fournit une bibliothèque `Java` effectuant l'analyse syntaxique (*parsing*) et le typage de programme `While`.

Le langage `While` sera décrit au fil des questions.

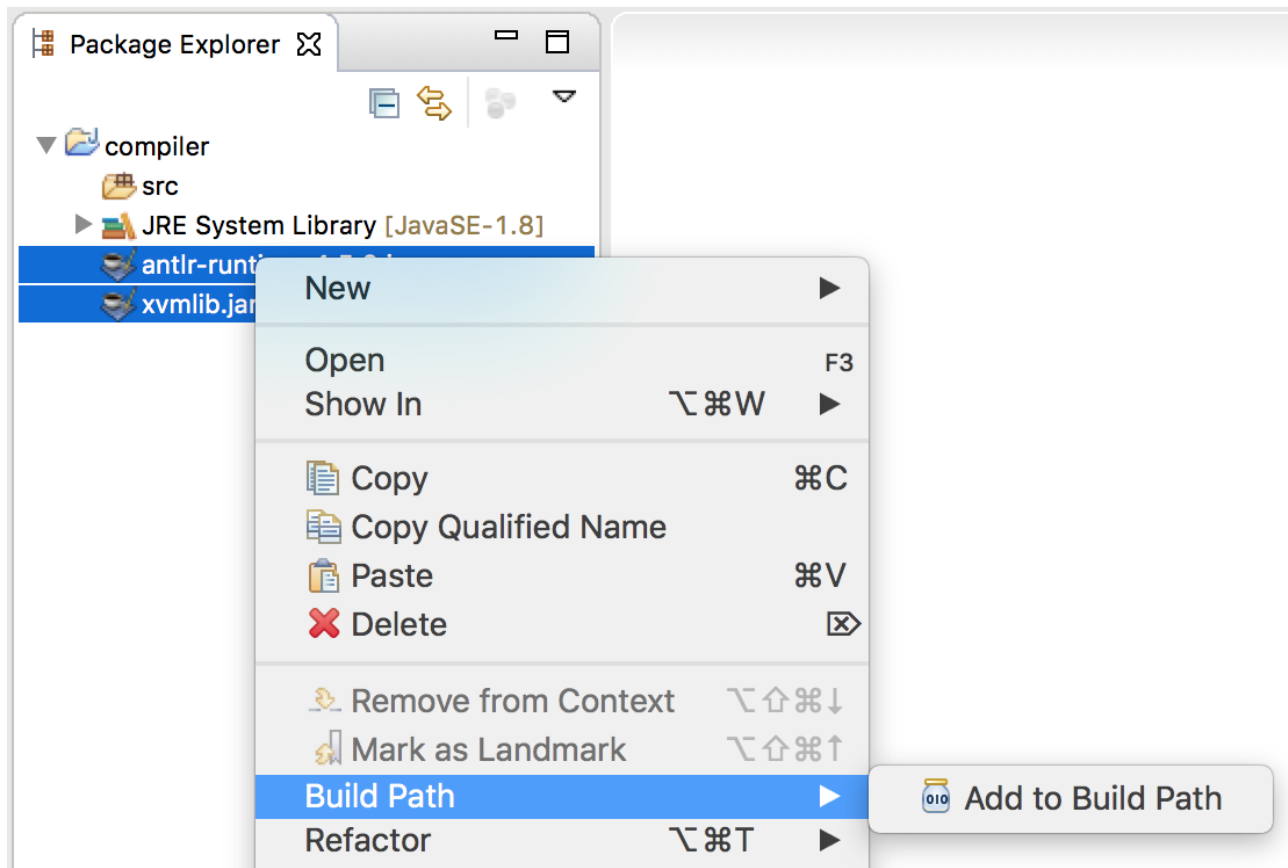
## Installation de la bibliothèque de support

Nous allons commencer par installer et utiliser la bibliothèque de support pour ce projet. Créez un nouveau projet `Java`, en refusant la création de `module-info.java` si Eclipse pose la question. Téléchargez les bibliothèques suivantes :

- [antlr-runtime-4.7.jar \(resources/lib/antlr-runtime-4.7.jar\)](#)
- [xvmlib.jar \(resources/lib/xvmlib.jar\)](#)

Avec VSCode, le plus simple est de copier ces 2 fichiers dans le répertoire `lib/` de votre projet (à côté de `src`).

Avec Eclipse, sauvez-les (e.g. en les glissant-déposant) à la racine du projet. Ensuite, sélectionnez-les dans l'explorateur d'Eclipse (fenêtre de gauche dans la vue par défaut - si elles n'apparaissent pas, rafraîchissez l'explorateur avec F5). Puis, faites un clic droit dessus et sélectionnez `Build Path -> Add to Build Path`, comme illustré ci-dessous :



Si, lors de la création du projet Java, vous avez laissé Eclipse créer un `module-info.java`, Eclipse risque d’interpréter cette opération différemment. Si vous avez des problèmes par la suite, revenez à `Build Path -> Configure Build Path`, cliquez sur `Libraries` en haut, et déplacez les deux `.jar` de `Modulepath` dans `Classpath`. Il pourra aussi être utile de supprimer `module-info.java`.

Ensuite, téléchargez l’archive suivante:

- [skeleton.zip \(resources/files/skeleton.zip\)](#).

et copier tous les fichiers qu’elle contient dans votre projet (i.e. les fichiers `*.java` dans le répertoire `src`). Ces fichiers constituent le squelette de votre compilateur que nous allons remplir par la suite.

Créez un fichier `example.wil` (vous pouvez créer un fichier via le menu `File -> New -> File`) à la racine de votre projet et copiez-y le programme `While` suivant:

```
int min(int a, int b) {
    if (a < b) return a; else return b;
}

void main() {
    print min(3, 5);
}
```

Enfin, lancez la fonction `main` de `TestParseAndType.java` et testez qu’il s’exécute sans erreur:

Ce code lit le contenu du fichier `example.wil`, puis en fait son analyse syntaxique et son typage en tant que programme `While`.

On vous fournit d’autres exemples de programmes `While`:

- [fact.wil \(resources/examples/fact.wil\)](#).
- [factrec.wil \(resources/examples/factrec.wil\)](#).
- [short-circuit.wil \(resources/examples/short-circuit.wil\)](#).

## Implémentation de référence

On vous fournit également une implémentation de référence:

- [mjava.jar \(resources/lib/mjava.jar\)](#).

Vous pouvez exécuter l’implémentation de référence via la commande suivante:

```
java -jar mjava.jar example.wil
```

Vous devriez voir s’afficher sur la sortie standard le code XVM produit par la compilation du fichier source en entrée.

Si vous le souhaitez, vous pouvez également exécuter l’implémentation de référence à partir d’Eclipse. Pour cela, créez un projet vide et importez les fichiers exemples ( `*.wil` ) ainsi que `mjava.jar` , puis ajoutez le `.jar` aux bibliothèques de votre projet (comme vous avez fait précédemment pour les 2 autres JAR). Ensuite, dans l’explorateur de `packages` d’Eclipse (sur la gauche), dépliez `mjava.jar/(default package)` pour y trouver `MJavaMain.class` . Faites un clic droit dessus, puis sélectionnez `Run as / Run configurations...` . Une boîte de dialogue apparaît. Après avoir vérifié que le programme sélectionné est bien `MJavaMain`, dans le tab `Arguments` , vous pouvez donner les arguments de ligne de commande du projet (ici, le nom du fichier source), puis enfin, cliquez sur `Run` . Ceci exécutera l’implémentation de référence.

## Concernant les tests automatisés

Les tests automatisés ne sont pas exhaustifs, il est important de tester vous-mêmes votre code. Vous pouvez utiliser la XVM de référence pour vérifier si le code assembleur généré produit bien l’effet voulu – lors d’un développement d’un projet, il est rare d’avoir un oracle omniprésent de correction et il est bien d’apprendre à coder sans.

Le code écrit pour des questions ultérieures peut introduire des dépendances entre fichiers, et vous forcer à revenir soumettre de nouveaux fichiers à la première question.

## Production d’un flux d’opcode

La bibliothèque `xvmlib` fournit une représentation des opcodes de la XVM (tels que `ADD` ou `GSB` ) et des étiquettes logiques de saut. Ces étiquettes représentent des adresses logiques de code, i.e. elles représentent une position nommée du code. Chaque instruction est représentée par une classe `Java` héritant de la classe abstraite `AsmInstruction` . Les étiquettes sont, quant à elles, directement représentées par le type `String` . Vous pouvez obtenir la description de toutes les instructions en consultant la [documentation de la bibliothèque](http://www.enseignement.polytechnique.fr/informatique/INF371/xvm-doc/doc/edu/polytechnique/xvm/asm/interfaces/AsmInstruction.html) (<http://www.enseignement.polytechnique.fr/informatique/INF371/xvm-doc/doc/edu/polytechnique/xvm/asm/interfaces/AsmInstruction.html>). Avec Eclipse ou VSCode, `ctrl+clic` sur un type amène à sa définition, ce qui peut aussi être utile pour comprendre comment utiliser ce type.

**Avertissement** : Vous pouvez voir dans la documentation que l’implémentation donnée et que vous allez utiliser propose une version de la XVM légèrement différente de celle du poly:

- Une instruction `LT` (qui teste l’inégalité stricte) à la place de `LEQ` (inégalité large).
- Une instruction `OR` (que vous n’êtes pas obligés d’utiliser mais qui peut être pratique - voir la remarque sur l’évaluation des expressions booléennes dans la partie suivante).

Nous allons commencer par créer une classe permettant de créer un flux d’opcode et de placer en son sein des étiquettes logiques. Ouvrez la classe `CodeGen` .

Cette dernière contient trois champs: `instructions` qui stocke la liste d’instructions en cours de construction, `labels` qui est un dictionnaire associant à chaque étiquette sa position dans la liste des instructions et `offsets` qui stocke les offsets (par rapport au Frame Pointer) des variables.

1. Implémentez les méthodes `generateLabel` , `pushLabel` , `pushInstruction` et `pushLocalVariable` :
  - la méthode `generateLabel()` doit générer et retourner une étiquette de code (ou *label*) frais - c’est-à-dire qui n’a jamais été utilisé auparavant,
  - la méthode `pushLabel(String label)` pousse l’étiquette `label` dans le dictionnaire `labels` . L’index associée à `label` devra pointer juste derrière la dernière instruction de la liste `instructions` au moment de l’insertion,
  - la méthode `pushInstruction(AsmInstruction asm)` poussera une nouvelle instruction dans le vecteur `instructions` , et

- la méthode `pushLocalVariable(String name, int offset)` associera à la variable `name` son `offset` par rapport au Frame Pointer.
2. La classe `TestCodeGen` contient une fonction `main` qui crée un programme `XVM` et affiche son code source en console. Utilisez la `XVM` en ligne pour exécuter le programme. Que fait ce dernier ?

Last submission: May 30, 2023 (16:55:31)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: CodeGen.java

## Compilation des expressions

À l’instar de Java, Le langage `While` est un langage typé statiquement et possède seulement deux types de base: `int` pour les entiers machines et `bool` pour les booléens.

Les expressions du langage sont construites à partir des variables , des constantes entières (en base 10), des valeurs booléennes `true` et `false` , ainsi que par application d’opérateurs built-in (telles l’addition ou la négation logique). (Il existe également une expression pour les appels de procédures que nous ignorons pour le moment)

Nous allons maintenant nous intéresser à la compilation de ces dernières. Le résultat d’une telle compilation sera un programme `XVM` qui évalue ladite expression et pousse le résultat de cette évaluation en tête de pile.

Les expressions sont représentées par la classe abstraite `AbstractExpr` , chaque construction étant matérialisée par une sous-classe concrète de `AbstractExpr` .

À l’exclusion des appels de procédures, il y a 5 types d’expressions: `EBool` , `EInt` , `EVar` , `EUniOp` et `EBinOp` qui représentent respectivement les littéraux booléens, les littéraux entiers, les variables, les opérateurs arithmético-logique unaires et les opérateurs arithmético-logique binaires. La méthode en charge de procéder à la compilation des expressions est `codegen(CodeGen cg)` . Elle est déclarée abstraite dans `AbstractExpr` et devra être implémentée dans toutes ses classes filles.

3. Implémentez les méthodes `codegen(CodeGen e)` dans toutes les classes filles de `AbstractExpr` (à l’exception de `ECall` ).

En exemple, on vous fournit l’implémentation de la méthode `EUniOp.codegen` .

Pour `EVar.codegen` , on s’autorisera à ajouter une méthode à la classe `CodeGen` permettant d’obtenir l’offset d’une variable.

On ne vous demande pas d’implémenter le *short-circuiting* des opérateurs booléens (e.g., l’assemblage de `e1 && e2` évaluera toujours les deux opérandes `e1` et `e2` ).

4. La classe `TestExprCodeGen` contient une fonction `main` qui crée un programme `XVM` à partir d’une expression et affiche son code source en console. (Le dictionnaire des positions des variables locales est construit pour vous, et utilise le fait qu’au démarrage la `XVM` positionne les registers FP/SP à `0` ) Utilisez ce code pour produire un programme `XVM` et utilisez la `XVM` en ligne pour l’exécuter. Vous pouvez librement modifier la variable `DEFAULT` pour tester d’autres expressions. Les affectations entre `<` `>` permettent de déclarer et d’initialiser les variables des l’expression.

Last submission: May 30, 2023 (16:55:13)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

## Compilation des instructions

Une instruction peut être une affectation, une conditionnelle, une boucle while ou une sortie de procédure. Nous donnons leur syntaxe respective ci-dessous, où `$block` désigne un bloc d'instructions (c'est-à-dire soit une instruction, soit une séquence d'instructions entre accolades):

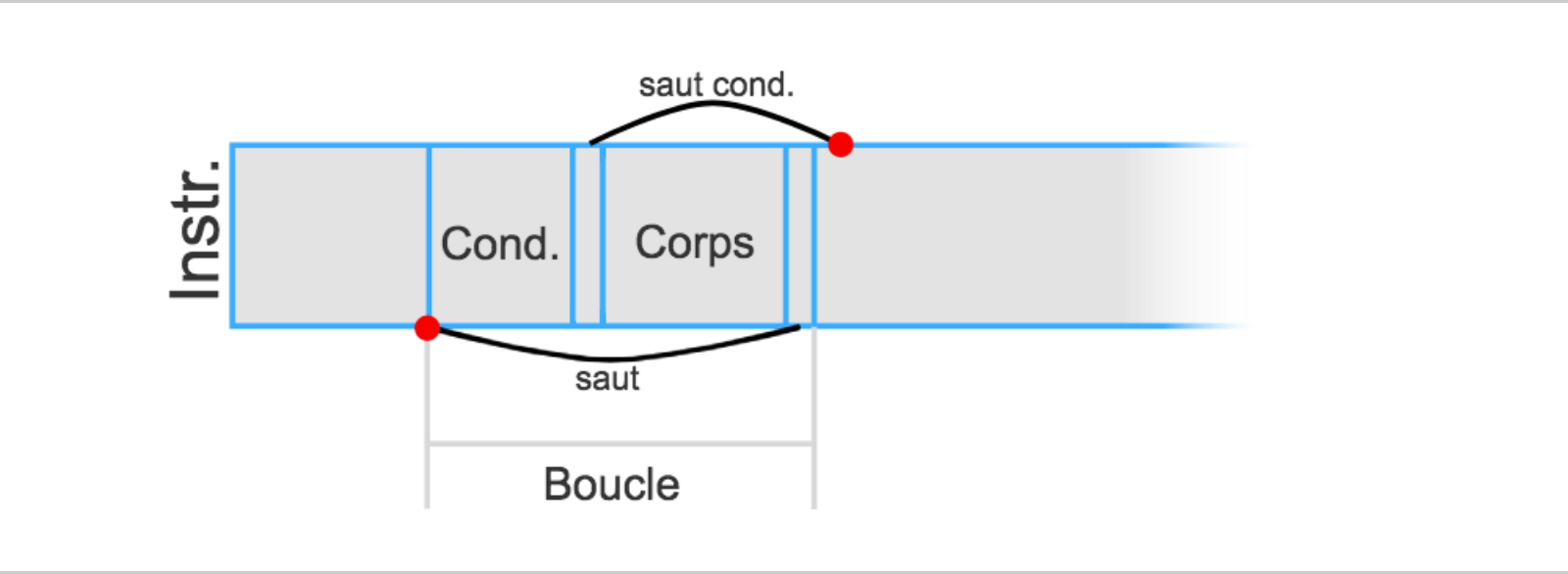
Instruction	Syntaxe
Affectation	<code>x = e;</code>
Evaluation d'une expression	<code>e;</code>
Conditionnelle	<code>if (e) \$block else \$block</code>
Conditionnelle (sans <code>else</code> )	<code>if (e) \$block</code>
Boucle while	<code>while (e) \$block</code>
Sortie de procédure	<code>return e;</code>
Sortie de procédure - pas de valeur de retour	<code>return ;</code>
Affichage d'une valeur (+ retour à la ligne)	<code>print e;</code>

Dans cette question, nous allons nous intéresser à la compilation des instructions - à l'exception de l'instruction `return`. Le résultat de la compilation sera un programme XVM qui évalue ladite instruction.

Les instructions sont représentées par la classe abstraite `AbstractInstruction`, chaque construction étant matérialisée par une sous-classe concrète de `AbstractInstruction`.

Les instructions qui nous intéressent dans cette question sont `IBlock`, `IAssign`, `IIf`, `IWhile` et `IPrint` qui représentent respectivement les blocs d'instructions, les affectations de variables, les conditionnelles, les boucles et l'instruction d'affichage. Par convention, une simple évaluation d'expression `e;` sera traitée comme une affectation sans rien à gauche du `=`.

Ici, la principale difficulté est la compilation du flot de contrôle. En effet, certaines instructions, telles la conditionnelle ou la boucle while, cassent la linéarité de l'exécution. Par exemple, si la condition d'une boucle while est fausse, le flot doit passer à la première instruction *après* la boucle, ce qui s'effectuera par un saut via les instructions `GTZ` (saut conditionnel) et `GTO` (saut non-conditionnel). Nous donnons ci-dessous une représentation graphique d'assemblage de la boucle while:



Nous voyons que le résultat de l'assemblage de la condition et du corps de la boucle sont côte à côte dans le flot d'instructions. Cependant, des instructions de saut ont été intercalées: une après l'évaluation de la condition et qui fait un saut si cette dernière est fausse; et une à la fin du corps de la boucle qui permet de revenir à l'évaluation de la condition.

La méthode en charge de procéder à la compilation des instructions est `codegen(CoGen cg)`. Elle est déclarée abstraite dans `AbstractInstruction` et devra être implémentée dans toutes ses classes filles.

5. Implémentez les méthodes `codegen(CoGen e)` dans toutes les classes filles de `AbstractInstruction` (à l'exception de `IReturn`).

En exemple, on vous fournit l'implémentation (partielle) de la méthode `IWhile.codegen`.

6. La classe `TestInstrCodeGen` contient une fonction `main` qui créé un programme `XVM` à partir d’une instruction et affiche son code source en console. (Le dictionnaire des positions des variables locales est construit pour vous, et utilise le fait qu’au démarrage la `XVM` positionne les registers FP/SP à `0` ) Utilisez ce code pour produire un programme `XVM` et utilisez la `XVM` en ligne pour l’exécuter. Vous pouvez librement modifier la variable `DEFAULT` pour tester d’autres instructions. Les affectations entre `<` `>` permettent de déclarer et d’initialiser les variables des expressions.

Last submission: May 30, 2023 (16:54:13)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: AbstractExpr.java, AbstractInstruction.java, CodeGen.java, EBinOp.java, EBool.java, EInt.java, EUniOp.java, EVar.java, IAssign.java, IBlock.java, IIf.java, IPrint.java, IWhile.java

## Procédures

Il est possible de définir des procédures en `While` . Une procédure peut prendre des arguments et peut retourner un résultat. Le corps d’une procédure contient une partie de déclaration de variables (avec optionnellement des valeurs d’initialisation), ainsi que le corps de la procédure sous la forme d’une séquence d’instructions:

```
rty name(ty1 arg1, ..., tyn argn) {
  tyx1 x1 = e1;
  ...
  tyxk xk = ek;

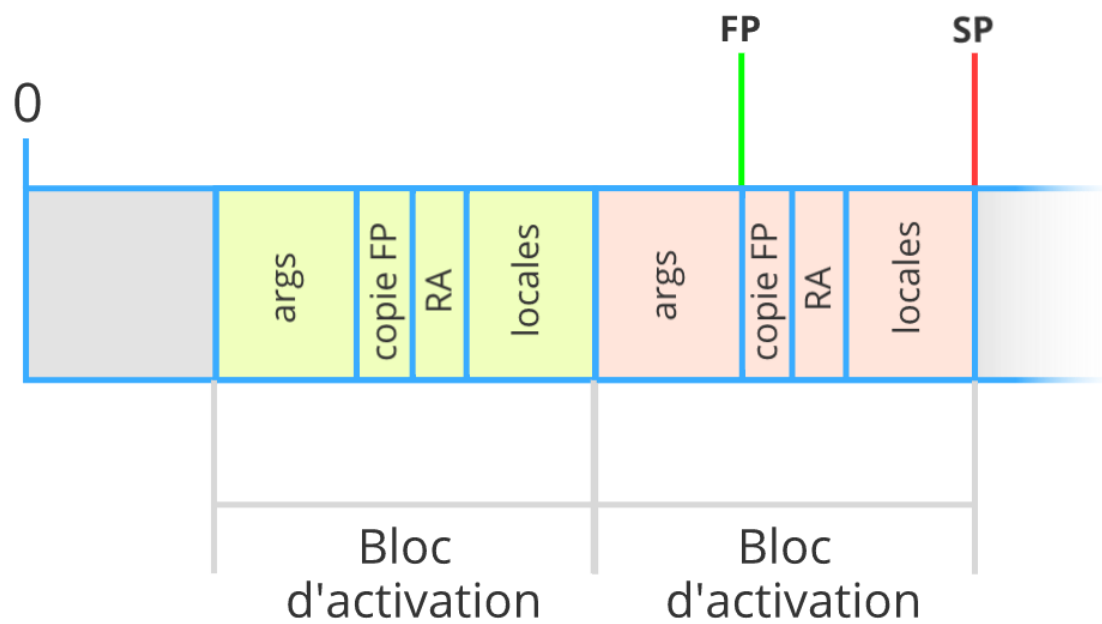
  $body;
}
```

Dans l’exemple ci-dessus, `name` est le nom de la procédure et `rty` est son type de retour (ou `void` si la procédure ne retourne pas de résultat). Cette procédure prend  $n$  arguments respectivement de type `ty1` , ..., `tyn` et qui sont nommés `arg1` , ..., `argn` . Enfin, la procédure déclare  $k$  variables locales nommées `x1` , ..., `xk` , respectivement de type `tyx1` , ..., `tyxk` et initialisées avec les expressions `e1` , ..., `ek` - la partie initialisation étant optionnelle.

Les procédures peuvent être mutuellement récursives.

Nous allons maintenant procéder à l’assemblage des procédures et de leurs appels. Pour cela, il faut décider de comment les arguments, variables locales, adresses de retour, etc... (ce qu’on appelle *bloc d’activation*) sont stockés et qui est en charge de construire et de nettoyer les différentes parties de ces blocs d’activation. L’ensemble de ces règles, qui varient d’une architecture et d’un langage à l’autre, est ce qu’on appelle la *convention d’appel*.

Nous allons maintenant définir la convention d’appel pour le langage `While` . Le bloc d’activation d’une procédure `While` est illustré sur le schéma ci-dessous où les deux derniers blocs d’activation actifs sont dessinés respectivement en vert et saumon:



On y voit que la pile sert à stocker plusieurs éléments: les arguments de la procédure, l’adresse de retour (qui servira quand on sortira de la procédure), une copie du registre FP de l’ancienne procédure et les variables locales. Il y a autant de blocs d’activation qu’il y a de procédures en cours d’exécution. Le dernier bloc d’activation est celui de la procédure active (i.e. de la procédure dernièrement appelée).

La convention d’appel est la suivante:

- L’appelant pousse sur la pile l’ensemble des arguments de la procédure de gauche à droite.
- Ensuite, il pousse sur la pile, dans cet ordre, une copie du registre FP et la valeur courante du pointeur d’instruction. Ensuite, il positionne respectivement les registres FP & SP sur la valeur initiale de SP et l’adresse de code de la procédure. L’ensemble de ces opérations peut être fait via l’instruction `GSB`.
- L’appelé commence son exécution et crée de l’espace sur la pile pour ses variables locales. Il a le droit de modifier le registre général `R`.
- Une fois l’exécution terminée, l’appelé utilise le registre `R` pour retourner son résultat et enlève de la pile ses variables locales, puis saute à l’adresse de retour qui est maintenant en tête de pile et restaure la valeur de FP — ces deux valeurs, qui ont été sauvegardées par l’appelant en haut de pile, doivent être nettoyées. Ces dernières opérations peuvent être faites via l’instruction `RET`.
- Finalement, l’appelant nettoie sa pile en y supprimant les arguments qu’il avait poussés initialement.

L’ajout des appels de procédures se fera à deux endroits. Dans l’assembleur pour les expressions (pour gérer la construction `ECall`) et dans l’assembleur des instructions (pour gérer la construction `IReturn`).

7. Complétez l’implémentation de `ECall.codegen(CoGen cg)`. Cette dernière implémentera la partie assemblage des *appels de procédures* (c’est-à-dire ce qui se passe dans l’appelant), la méthode statique `ProgramCodeGen.labelOfProcName(name)` vous donnant le nom d’étiquette où il faudra sauter pour commencer à exécuter la procédure `name`.
8. Complétez l’implémentation de `IReturn.codegen(CoGen cg)`. Cette dernière implémentera la partie assemblage des *retours/sorties des appels de procédures*.
9. Complétez l’implémentation de `ProgramCodeGen.codegen(TProcDef proc)` qui a en charge d’assembler une procédure dans son intégralité du point de vue de l’appelé (i.e. création des variables locales, puis exécutions du corps — le nettoyage de la pile étant fait par l’instruction `return`). Les instructions devront être poussées dans le `CoGen` stocké dans l’attribut `cg`. Il faudra bien sûr construire la map `offsets`. Pensez aussi à la gestion des fonctions ne se terminant pas par un `return` explicite.
10. Enfin, utilisez `TestProgramCodeGen` pour compiler différents programmes `While`. Testez les produits de la compilation grâce à la `XVM` en ligne.

Last submission: June 02, 2023 (01:50:59)

Choose

Choose one or more files...

Submit



☒ Reuse files from previous submissions 

Expected files: AbstractExpr.java, AbstractInstruction.java, CodeGen.java, EBinOp.java, EBool.java, ECall.java, EInt.java, EUniOp.java, EVar.java, IAssign.java, IBlock.java, IIf.java, IPrint.java, IReturn.java, IWhile.java, ProgramCodeGen.java

## Enregistrements extensibles

**Cette partie est optionnelle**

### Enregistrements en While

Nous allons étendre le langage `While` avec un système d'enregistrements (*record*) extensibles. Un enregistrement est un type composé qui contient d'autres variables (appelées *champs*) de noms et de types prédéfinis. Ils sont proches des classes `Java` sans méthode. En `While`, il est possible de déclarer de nouveaux enregistrements grâce au mot-clef `record`. Par exemple, le code suivant déclare un nouvel enregistrement `point` qui contient deux champs `x` et `y` de types `int`.

```
record point {  
    int x;  
    int y;  
}
```

A l'instar de `Java`, il n'est possible que de stocker des *références* vers les enregistrements, et il faut toujours les allouer dynamiquement via le mot-clef `new`. Par exemple:

```
point p = new point;
```

La valeur des champs d'un enregistrement fraîchement alloué est non spécifiée. Enfin, il est possible de lire/écrire la valeur d'un champ grâce à la notation pointée `var.field`. Par exemple:

```
point p = new point;  
int z;  
  
p.x = 0; p.y = 5; // Set  p.x && p.y  
z = p.x * p.y;    // Read p.x && p.y
```

Enfin, les enregistrements sont extensibles, i.e. qu'il est possible de créer de nouveaux types enregistrements par extension d'un ancien. Ceci se fait grâce au mot-clef `extends`, comme dans l'exemple suivant où les types `rectangle` et `circle` sont des extensions de `shape`, et possède donc un champ `point center` à ce titre:

```
record shape {  
    point center;  
}  
  
record rectangle extends shape {  
    int height;  
    int width;  
}  
  
record circle extends shape {  
    int radius;  
}
```

Tout enregistrement qui en étend un autre peut-être vu (via un *cast* ou transtypage) comme un enregistrement du type qu'il étend:

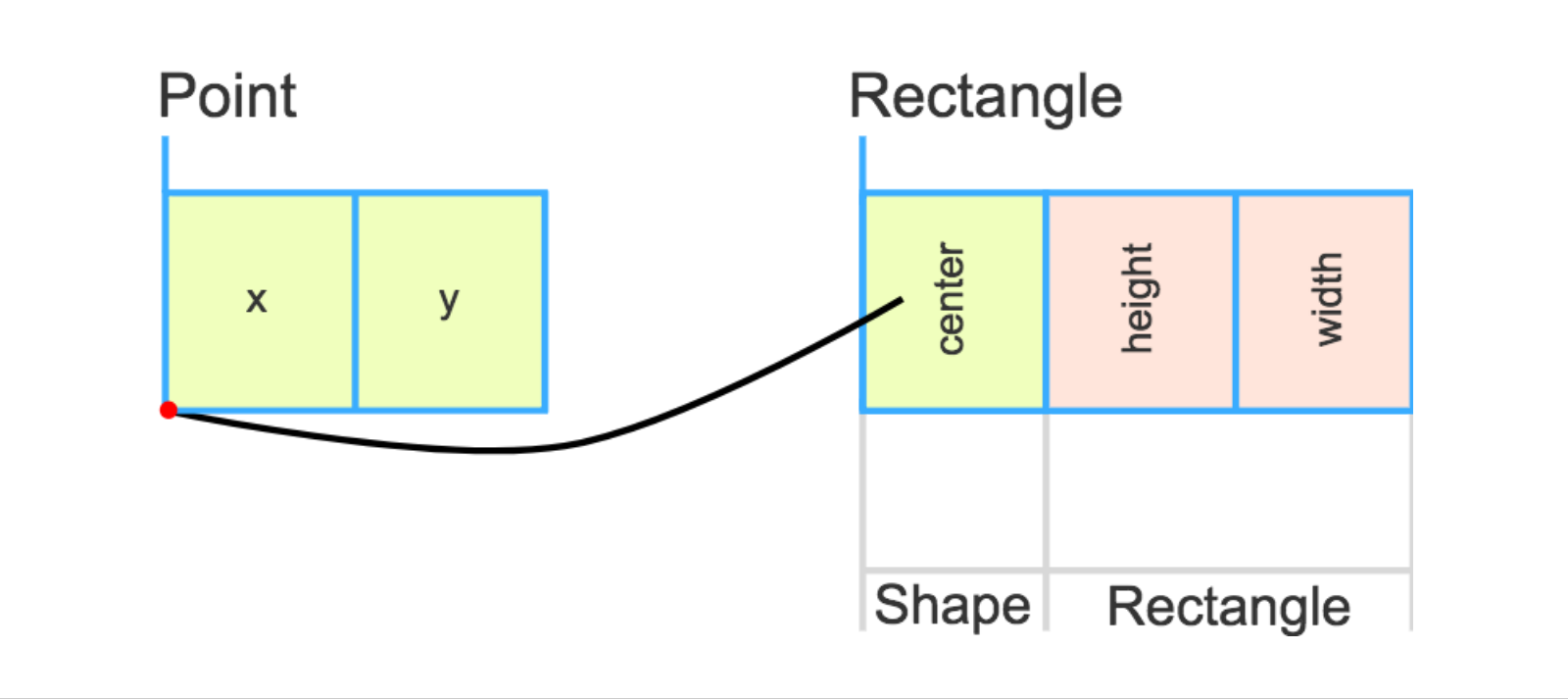


```
void foo(shape s) {
    // some code
}

void main() {
    var circle c = new circle;
    foo(s); // s is cast to `shape`
}
```

Représentation mémoire

Les enregistrements sont représentés en mémoire par un espace continu qui permet de stocker les champs dans leur ordre de déclaration. Dans le cas d’extension d’enregistrements, la représentation restera contiguë, et celle de l’enregistrement étendu précédera celle de l’extension. Par exemple:



Ainsi, la représentation mémoire de l’extension reste une représentation mémoire valide de l’étendu (au dessus, le champ `center` est à la même position pour `shape`, `rectangle` et `circle`), et l’opération de transtypage n’aura donc aucune incidence sur la représentation mémoire. Notez également que les champs de type enregistrement sont stockés par référence. (Ci-dessus, le champ `center` est une référence vers un `point` )

Intégration du nouveau typeur à votre projet

On s’intéresse maintenant à la partie assemblage des enregistrements. Les enregistrements apportent les modifications suivantes à l’arbre de syntaxe:

- l’introduction de 2 nouveaux noeuds *expression* : un pour l’allocation d’un enregistrement ( `ENew` ), un pour l’accès au champ d’un enregistrement ( `EGet` ).
- la classe `IAssign` est modifiée afin de prendre en compte qu’une valeur gauche (i.e. la destination d’une affectation) peut être le champ d’un enregistrement.
- la fonction de typage ne renvoie plus seulement une liste de procédure mais également une liste de définitions d’enregistrements.

Pour démarrer, effectuez les modifications suivantes à votre projet:

- Téléchargez l’archive [skeleton-x.zip](#) ([resources/files/skeleton-x.zip](#)) et ajoutez son contenu à votre projet. Cette dernière contient les noeuds `EGet` et `ENew`, une classe `RecordsInfo` qui servira à stocker des informations précalculées sur les enregistrement et un program de test `TestProgramCodeGenWithRecords` .
- Modifiez la classe `IAssign` en changeant le type du champ

```
public final Optional<String> lvalue;
```

en

```
import edu.polytechnique.mjava.ast.LValue;

// ...

public final Optional<LValue<AbstractExpr>> lvalue;
```

La classe `LValue` contient 2 champs:

- `Optional<TypedExpr> target` qui, quand il est présent, est l'expression qui représente l'enregistrement qui sera modifié, et
- `String name` qui contient le nom du champ ou de la variable qui sera affecté.

Remplacez les constructeurs de `IAssign` par les 2 suivants:

```
public IAssign(Optional<String> lvalue, AbstractExpr rvalue) {
    this.lvalue = lvalue.map((x) -> new LValue<AbstractExpr>(x));
    this.rvalue = rvalue;
}

public IAssign(LValue<AbstractExpr> lvalue, AbstractExpr rvalue) {
    this.lvalue = Optional.of(lvalue);
    this.rvalue = rvalue;
}
```

- Réparez l'implémentation de `IAssign.codegen`. A ce stade, vous ignorez le champ `target` de `lvalue`.

À ce state, votre compilateur doit de nouveau être opérationnel.

## Travail demandé

On s'intéresse maintenant à la partie assemblage des enregistrements, la partie parseur / analyseur de type fournie en début de TD supportant déjà ces derniers. Notamment, le type `TProgram` possède un champ `records` qui contient une liste de tous les enregistrements définis.

11. La classe `RecordsInfo` va nous servir à stocker, pour chaque enregistrement, sa taille en mémoire ainsi que le décalage auquel sera stocké chaque champ (y compris les champs hérités par extension).

On vous demande de compléter la méthode `visit1` qui calcule le `RecordsInfo` pour l'enregistrement `ty` passé en paramètre et stocke le résultat dans le dictionnaire `infos`. Le premier argument `mtypes` est un dictionnaire qui contient toutes les définitions de type du programme. Notez qu'il n'est pas garanti que le `RecordsInfo` du parent ait été calculé avant celui de son enfant.

12. Modifiez la classe `CodeGen` de manière à ce que cette dernière possède un champ

```
Map<String, RecordsInfo> types;
```

qui stockera toutes les informations (taille, décalages des champs) sur les enregistrements. Modifiez son constructeur en conséquence (en créant une map vide) et ajoutez le constructeur suivant:

```
public CodeGen(Map<String, RecordsInfo> types) {
    this();
    this.types = types;
}
```

Adaptez votre code ( `ProgramCodeGen` et `TestProgramCodeGenWithRecords` ) en conséquence — notamment pour faire quelque chose du champ `records` de la classe `TProgram` dans `TestProgramCodeGenWithRecords` pour transmettre le dictionnaire `types` à `ProgramCodeGen`.


13. Complétez la méthode `codegen` des classes `ENew` et `EGet` qui respectivement alloue un nouvel enregistrement et lit un champ dans une expression de type enregistrement.
14. Modifiez la méthode `codegen` de `IAssign` afin que cette dernière prenne en charge l'écriture d'un champ d'un enregistrement.

15. Testez votre implémentation (on vous fournit un exemple ([resources/examples/records.wil](#))).

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions 

Expected files: AbstractExpr.java, AbstractInstruction.java, CodeGen.java, EBinOp.java, EBool.java, ECall.java, EGet.java, EInt.java, ENew.java, EUniOp.java, EVar.java, IAssign.java, IBlock.java, IIf.java, IPrint.java, IReturn.java, IWhile.java, ProgramCodeGen.java, RecordsInfo.java

## Optimisation des appels terminaux


**Cette partie est optionnelle**

16. Optimisez l’implémentation des appels terminaux lorsque la fonction appelée n’a pas plus d’arguments que la fonction appelante. On pourra se limiter au cas de `return fun(...)`; et ignorer le cas où la procédure appelante ne renvoie pas de résultat, où l’identification des appels terminaux serait plus complexe.

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions 

Expected files: AbstractExpr.java, AbstractInstruction.java, CodeGen.java, EBinOp.java, EBool.java, ECall.java, EInt.java, EUniOp.java, EVar.java, IAssign.java, IBlock.java, IIf.java, IPrint.java, IReturn.java, IWhile.java, ProgramCodeGen.java