

Langages, interprétation, compilation

DM - Kawa

L'objectif de ce DM est de construire un interprète pour un petit langage objet inspiré de Java.

Description du langage Kawa

Un programme Kawa est formé par une série de déclarations de variables globales, suivie d'une série de définitions de classes, suivie d'un bloc d'instruction principal à exécuter.

La déclaration d'une variable mentionne son type et son nom.

```
var int n;  
var point p;
```

Un type peut être un type de base `int` ou `bool`, ou un nom de classe. La définition d'une classe comporte d'abord des déclarations d'attributs, typés comme les variables, puis des définitions de méthodes. Chaque définition de méthode comporte un type de retour (ou la mention spéciale `void` si la méthode ne renvoie rien), une liste de paramètres et leurs types, et une séquence d'instructions à exécuter.

```
class point {  
    attribute int x;  
    attribute int y;  
  
    method void constructor(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    method int size() { return this.x+this.y; }  
}
```

Le bloc de code principal est placé entre accolades, après le mot-clé `main`.

```
main {  
    p = new point(1, 2);  
    p.x = p.x + p.y;  
    print(p.size());  
}
```

La construction d'un nouvel objet se fait avec l'opérateur `new` suivi d'un nom de classe, et éventuellement d'une séquence d'arguments entre parenthèses. Par défaut, les attributs de l'objet créé ne sont pas initialisés. Lorsque l'on fournit une liste d'arguments (fût-elle vide), la création de l'objet est suivie de l'appel de la méthode `constructor` de la classe concernée avec les arguments fournis, sur l'objet nouvellement créé. Note : dans ce cas, la méthode `constructor` doit exister, et son retour doit être `void`.

Travail à réaliser (base)

Le DM est découpé en trois parties :

1. **analyse syntaxique**,
2. **vérification des types**,
3. **interprétation**,

auxquelles s'ajoutent des **extensions** optionnelles.

Consignes

On fournit un **squelette de code** contenant les éléments suivants.

Fichier	Contenu	Commentaire
<code>kawa.ml</code>	syntaxe abstraite	
<code>kawalexer.mll</code>	analyse lexicale	à compléter (ocamllex)
<code>kawaparser.mly</code>	analyse grammaticale	à compléter (menhir)
<code>typechecker.ml</code>	vérification des types	à compléter (caml)
<code>interpreter.ml</code>	interprétation	à compléter (caml)
<code>kawai.ml</code>	programme principal	
<code>dune/dune-project</code>	configuration	
<code>tests</code>	dossier de tests	à compléter (kawa)

Votre travail principal consiste à compléter les quatre fichiers `kawalexer.mll`, `kawaparser.mly`, `typechecker.ml` et `interpreter.ml` en respectant les descriptions données dans la suite de cette page. Vous obtiendrez alors avec le programme `kawai` un interprète complet pour `Kawa`. On s'attend à ce que vous ajoutiez également de nouveaux tests.

Vous pouvez inclure à votre projet autant d'extensions que vous le souhaitez. Les extensions sont facultatives, mais si vous en réalisez elles viendront renforcer votre note.

Vous pouvez travailler seul ou en binôme. Votre projet devra être rendu à votre encadrant de TP. Joignez à votre projet un rapport décrivant ce qui a été réalisé, ce qui fonctionne ou non, et les difficultés que vous avez pu rencontrer. Le rapport doit également détailler les éventuelles extensions que vous avez traitées. Le rapport peut prendre la forme d'un simple fichier `README.txt`.

Le squelette de code est conçu pour traiter correctement un programme minimal. En plus de ces éléments, vous pouvez réutiliser sans limitations les fragments de code présentés dans le cours. Attention en revanche : tout emprunt de code d'une autre source que le cours doit être documenté dans votre rapport.

Recommandations

Votre progression dans ce projet peut se représenter sur deux axes :

1. les étapes successives du programme d'interprétation, qui correspondent aux différents fichiers à compléter,
2. les différentes constructions du langage `Kawa`, qui correspondent aux différents cas de chaque fonction principale.

Vous pouvez vous figurer un tableau à double entrée comme le suivant.

	<code>kawalexer</code>	<code>kawaparser</code>	<code>typechecker</code>	<code>interpreter</code>
Arithmétique				
Variables				
Instructions				
Classes et attributs				
Méthodes				
Héritage				

Vous pouvez organiser votre travail selon l'un ou l'autre de ces axes.

- En travaillant colonne par colonne, vous suivez les chapitres du cours (et les sections de ce sujet), et vous complétez un fichier avant de commencer le suivant.
- En travaillant ligne par ligne, vous vous concentrez sur les différents aspects du langage l'un après l'autre, et progressez en parallèle dans les quatre fichiers.

La deuxième stratégie a un avantage : elle permet de réaliser rapidement des tests, en sélectionnant des programmes `Kawa` contenant exactement les aspects traités.

Remarquez que dans l'étape intermédiaire où on a les classes et les attributs, mais pas encore les méthodes, le langage ne permet de créer un objet que d'une seule manière : avec un nom de classe mais pas d'arguments.

```
p = new point;
```

Les attributs sont alors systématiquement non initialisés. Une fois les méthodes ajoutées en revanche,

on peut inclure la création d'objets initialisés par un constructeur.

Quelle que soit votre stratégie de progression, il est important de tester votre programme à chaque étape avant de passer à la suivante.

- Le programme principal `kawai` est un interprète `Kawa`. Il prend en entrée un nom de fichier `.kwa`, fait son analyse syntaxique, vérifie sa cohérence de types, et l'interprète.
- Le dossier `tests` contient quelques tests, à compléter par les vôtres.

L'ensemble du projet peut être résolu en 400 lignes de code environ, avec quatre fichiers de tailles comparables.

1. Analyse syntaxique

La première tâche consiste à réaliser l'analyse syntaxique d'un programme et à produire l'arbre de syntaxe abstraite associé. Vous devez pour cela compléter les fichiers `kawalexer.mll` et `kawaparser.mly`. Vous devez aussi expurger votre analyseur syntaxique de toutes ses ambiguïtés (à l'aide de priorités).

Syntaxe concrète

La syntaxe concrète de `Kawa` est définie par les règles suivantes (des explications de certaines notations viennent juste après).

```

<program> ::= <var_decl>* <class_def>* <main> eof

<class_def> ::= class ident [extends ident]? { <attr_decl>* <method_def>* }

<var_decl> ::= var <type> ident ;
<attr_decl> ::= attribute <type> ident ;

<type> ::= int
        | bool
        | ident
        | void

<method_def> ::= method <type> ident ( [<type> ident /,]* ) { <var_decl>* <instr>*

<expr> ::= n
        | true
        | false
        | this
        | <mem>
        | <uop> <expr>
        | <expr> <bop> <expr>
        | ( <expr> )
        | new ident
        | new ident ( [<expr> /,]* )
        | <expr> . ident ( [<expr> /,]* )

<mem> ::= ident
        | <expr> . ident

<instr> ::= print ( <expr> ) ;
        | <mem> = <expr> ;
        | if ( <expr> ) { <instr>* } else { <instr>* }
        | while ( <expr> ) { <instr>* }
        | return <expr> ;
        | <expr> ;

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

```

On prend comme symboles terminaux :

- les constantes entières positives, désignées dans les règles par `n`,
- les identifiants alpha-numériques, désignés dans les règles par `ident`,
- les mots-clés `true`, `false`, `var`, `attribute`, `method`, `class`, `new`, `this`, `if`, `else`, `while`, `return`, `print`, `int`, `bool`, `void`,
- les symboles `=`, `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `<=`, `&&`, `||`, `(`, `)`, `{`, `}`, `;`, `.`, `,`,
- un symbole spécial `eof` (fin de fichier)

On prend comme symboles non terminaux principaux :

- `<program>` pour un programme complet
- `<class_def>` et `<method_def>` les définitions de classes et de méthodes
- `<var_decl>` et `<attribute_decl>` pour les déclarations de variables et d'attributs
- `<type>` pour un type
- `<expr>` pour une expression
- `<mem>` pour un accès mémoire
- `<instr>` pour une instruction
- `<uop>` et `<bop>` pour les opérateurs unaires et binaires

Vous pourrez en introduire d'autres en fonction des besoins, pour reproduire les différents motifs de la grammaire. Rappel : la notion d'« accès mémoire » sert ici à regrouper les accès à une variable ou à un attribut d'un objet (que ce soit pour lecture ou pour écriture).

Dans les règles de grammaire, on adopte les notations supplémentaires suivantes :

- `[m]?` pour désigner la présence optionnelle du motif `m`
- `[m]*` pour désigner une répétition éventuellement vide du motif `m`
- `[m/,]*` pour désigner une répétition éventuellement vide du motif `m`, où les occurrences sont séparées par `,`

En outre, on autorise des commentaires (non imbriqués) délimités par les séquences `/*` et `*/`, ou par `//` et une fin de ligne.

On conserve toutes les conventions d'écriture et priorités des opérateurs en vigueur dans les langages usuels (par exemple, en java).

Syntaxe abstraite

La syntaxe abstraite retranscrit directement la plupart des constructions de la syntaxe concrète. Elle est principalement définie par les types suivants :

- `program` pour les programmes complets
- `class_def` et `method_def` pour les définitions de classes et de méthodes
- `expr` et `mem_access` pour les expressions et accès
- `instr` et `seq` pour les instructions et séquences
- `typ` pour les types

Apparaissent également des types `uop` et `bop` qui énumèrent les opérateurs.

2. Vérification des types

Patience, on en parlera bientôt en cours.

3. Interprétation

L'interprétation d'un programme `Kawa` doit exécuter la séquence d'instructions principale. À mesure de cette exécution, il faudra créer ou modifier des objets en mémoire, et afficher des valeurs.

L'évaluation d'une expression produite une valeur de l'une des quatre sortes suivantes :

- constante entière,

- constante booléenne,
- constante `null` (lorsqu'une valeur est indéfinie),
- objet.

Un objet est composé d'un nom de classe, et d'une table donnant une valeur (éventuellement `null`) à chaque attribut.

Mise en place

Le fichier `interpreter.ml` contient un fragment d'interprète, que vous devez compléter. Vous y trouverez en particulier :

- une fonction `exec_prog`, qui prend en paramètre un programme `Kawa` et l'exécute,
- une fonction `eval_seq`, qui exécute une séquence d'instructions,
- une fonction `eval_call`, qui évalue un appel de méthode.

Les fonctions `eval_seq` et `eval_call` accèdent toutes les deux à un environnement global `env` mutable, qui associe une valeur à chaque variable globale et est mis à jour lors des opérations d'affectation. En outre, `eval_seq` utilise un environnement local `lenv`, mutable également, qui associe une valeur à chaque paramètre de fonction et chaque variable locale de l'appel en cours.

Le fragment de code fourni fait les choix techniques suivants.

- L'environnement local et l'environnement global sont réalisés par des tables de hachage, dont les clés sont les identifiants des variables et des paramètres. À chaque clé, la table associe une valeur, qui est `null` pour une variable non initialisée.
- Chaque objet est représenté un record caml avec deux champs : le premier est le nom de la classe à laquelle appartient l'objet, le deuxième est une table de hachage dont les clés sont les noms des attributs, et qui donne la valeur associée à chacun.

Vous avez le droit de modifier ces choix techniques si vous préférez un style différent.

Sémantique

L'interprétation doit suivre les principes suivants.

Valeurs et opérations de base.

- Les constantes, entières ou booléennes, sont leur propre valeur.
- La valeur `null` caractérise les variables ou attributs non initialisés, et le résultat d'une fonction `void`.
- Les opérations et comparaisons arithmétiques ont leur signification habituelle. L'ordre d'évaluation des opérandes d'une opération binaire n'est généralement pas spécifié.
- Rappel : les opérations `&&` et `||` sont paresseuses. Elles évaluent en premier leur opérande de gauche, et n'évaluent celui de droite que si nécessaire.
- Les opérations `==` et `!=` testent l'égalité physique de leurs opérandes. Elles obéissent aux critères suivants :
 - chaque constante est égale à elle-même,
 - deux objets sont égaux si et seulement s'ils sont physiquement le même objet,
 - des valeurs de natures différentes ne sont jamais égales.

Classes et objets.

- Une définition de classe définit un type et des méthodes. L'ensemble des classes définies dans un programme est inclus dans le contexte de l'interprète.
- Chaque classe hérite optionnellement d'une autre classe appelée son parent. Elle en reprend alors les déclarations d'attributs et les définitions de méthodes.
- L'opération `new` appliquée uniquement à un nom de classe `cn` crée un nouvel objet de la classe `cn`, et renvoie cet objet. Les attributs de l'instance créée ne sont pas initialisés (leur consultation doit renvoyer `null`).
- L'opération `new` appliquée à un nom de classe `cn` et une séquence de paramètres (`e1, ..., eK`) crée d'abord un nouvel objet de classe `cn` comme ci-dessus, puis appelle sur cet objet la méthode `constructor` de la classe `cn`. À nouveau, le résultat est l'instance créée.

Variables et attributs.

- Les variables sont mutables, de même que les attributs des objets.
- Une variable globale déclarée au début du programme est visible dans tout le code du programme. Une variable locale décrite au début d'une méthode est visible exclusivement dans le code de cette méthode, et y masque une éventuelle variable globale de même nom.
- Un accès `e.x` suppose que la valeur de `e` est un objet dont la classe (ou une classe parente) possède un attribut de nom `x`. Lors d'un accès en écriture, la valeur de cet attribut doit être modifiée.

Méthodes.

- L'évaluation d'un appel de méthode `e.f(e1, ..., eK)` suppose que la valeur de l'expression `e` est un objet, appartenant à une classe qui définit (ou hérite d')une méthode `f`. Pour choisir la méthode appelée, on cherche d'abord dans la classe de `e` elle-même, puis dans son éventuel parent, et ainsi de suite en remontant jusqu'à trouver une méthode du nom `f` demandé. C'est cette dernière qui est appelée, avec les paramètres explicites `e1` à `eK`. L'objet donné par `e` est également accessible durant l'évaluation de la méthode en tant que paramètre implicite (désigné par `this`).
- Dans un appel de méthode `e.f(e1, ..., eK)`, l'ordre d'évaluation des expressions `e`, `e1`, ..., `eK` n'est pas spécifié.

Extensions

Patience ici aussi (vous avez vraiment déjà fini le reste ?).