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'une 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
`kawa.ml`	syntaxe abstraite	
`kawalexer.mll`	analyse lexicale	à compléter (ocamllex)
`kawaparser.mly`	analyse grammaticale	à compléter (menhir)
`typechecker.ml`	vérification des types	à compléter (caml)
`interpreter.ml`	interprétation	à compléter (caml)
`kawai.ml`	programme principal	
`dune`/`dune-project`	configuration	
`tests`	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.

	`kawalexer`	`kawaparser`	`typechecker`	`interpreter`
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` contients 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

La vérification des types en Kawa doit vérifier en particulier que chaque expression est cohérente, que chaque accès à un champ et chaque appel de méthode est bien permis par la classe de l'objet concerné, et que chaque appel et chaque affectation est faite avec des expressions dont le type statique est bien un sous-type du type attendu. Elle doit produire une erreur dans (au moins) les cas suivants :

- expression incohérente,
- référence à une variable inexistante,
- référence à un champ d'une valeur qui n'est pas un objet,
- référence à un nom de champ inexistant,
- présence d'un `return` dans une fonction ne renvoyant pas de résultat.

Mise en place

Le fichier `typeckecker.ml` contient :

- une fonction `typecheck_prog : program -> unit` qui termine lorsque le programme donné en argument est correct, et qui produit une erreur sinon,
- deux fonctions `type_expr` et `type_mem_access` qui prennent qui prend en arguments une expression (resp. un accès mémoire) et un environnement de typage, et qui renvoient le type de l'expression ou de l'accès, ou produisent une erreur en cas d'incohérence,

- une fonction `check` qui prend en paramètre une expression, un type attendu et un environnement, et qui vérifie que l'expression est bien typée, et du type attendu, et produit une erreur sinon,
- deux fonctions `check_instr` et `check_seq`, qui prenennt en paramètre une instruction ou séquence, un type de retour attendu (`TVoid` si pas de retour attendu) et un environnement de typage, qui vérifie que l'instruction est cohérente et produit une erreur sinon.

Votre tâche consiste à compléter ces six fonctions, en respectant les règles de typage détaillées ci-dessous. Vous devrez également ajouter deux fonctions `check_class` et `check_mdef` pour typer respectivement les définitions de classes et de méthodes.

Notes supplémentaires sur les environnements de typage :

- lors du typage du code d'une classe `C`, on suppose que l'environnement de typage contient une variable spéciale `this` de type `C`,
- lors du typage du code d'une méthode, on suppose que l'environnement de typage contient des entrées pour les paramètres et les variables locales de cette méthode,
- toutes les fonctions ont accès à la liste des classes définies par le programme à analyser, et peuvent donc, connaissant un nom de classe, aller en extraire les informations de typage associées.

Syntaxe abstraite des types : le squelette de code contient la définition suivante pour la représentation des types.

```
type typ =
  | TVoid
  | TInt
  | TBool
  | TClass of string
```

Bien qu'on ne considère pas `void` comme un vrai type, on a inclus dans cette représentation un constructeur `TVoid`. Ce choix va simplifier l'écriture du vérificateur de types, en évitant notamment de traiter séparément le cas d'une méthode qui renvoie une valeur (d'un "vrai" type) et le cas d'une méthode qui ne produit pas de valeur (cas `void`). Alternativement, on aurait pu représenter `void` par `None`, et les vrais types par `Some ...`.

Règles de typage

Remarque : pour alléger l'écriture de ces règles, on y utilise autant que possible la syntaxe concrète.

On note

```
t' <: t
```

lorsque `t'` est un sous-type de `t`, c'est-à-dire lorsque

- `t'` est égal à `t`,
- ou `t'` est une sous-classe, directe ou indirecte, de `t`.

Constantes, où `n` désigne une constante entière.

```
------------    -----------------    ------------------
E |- n : int    E |- true : bool    E |- false : bool
```

Opérations unaires.

```
E |- e : int       E |- e : bool
-------------    ------------------
E |- -e : int    E |- not e : bool
```

Opérations binaires. Les opérations et comparaisons arithmétiques travaillent sur des entiers. Le test d'égalité est polymorphe mais demande des opérandes homogènes.

```
E |- e1 : int     E |- e2 : int      E |- e1 : int     E |- e2 : int
-------------------------------      -------------------------------
      E |- e1 + e2 : int                   E |- e1 < e2 : bool

E |- e1 : t     E |- e2 : t
```

```
---------------------------
   E |- e1 == e2 : bool
```

Variables.

```
 E(x) = t       E |- e : C     C(x) = t
----------      ----------------------
E |- x : t           E |- e.x : t
```

Appels de méthodes et de constructeurs (avec dans les prémisses, une vérification à faire pour tout `k`).

```
   C existe
--------------
E |- new C : C

C(constructor) = (t1 x ... x tN) -> void      E |- ek : tk'      tk' <: tk
-------------------------------------------------------------------------
                       E |- new C(e1, ..., eN) : C

E |- e : C     C(f) = (t1 x ... x tN) -> t      E |- ek : tk'      tk' <: tk
-------------------------------------------------------------------------
                       E |- e.f(e1, ..., eN) : t
```

À noter dans les deux dernières règles : en l'absence d'héritage la condition de sous-typage `tk' <: tk` est remplacée par un simple test d'égalité.

Instructions

```
 E |- e : int        E |- e : void       E |- e : t   t = ret
--------------      --------------      --------------------
E |- print(e);          E |- e;             E |- return e;

E |- e : t    t <: E(x)        E |- e1 : C      E |- e2 : t      t <: C(x)
-----------------------        ------------------------------------------
      E |- x = e;                          E |- e1.x = e2;

E |- e : bool     E |- s1     E |- s2       E |- e : bool     E |- s
-------------------------------------      ------------------------
   E |- if (e) { e1 } else { e2 }             E |- while (e) { s }
```

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 gloable 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 el 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 sont 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 paramètre `(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

Voici une liste de suggestions d'extensions, concernant différents aspect de votre interprète. Cette liste n'est pas limitative.

Enrichissements du langage

Champs immuables

Ajouter au langage la possibilité de déclarer un attribut `final`. La valeur d'un tel attribut ne peut pas être modifiée. Votre vérification des types devra donc s'assurer que seul le constructeur de la classe correspondante affecte une valeur à un tel attribut.

Visibilités

Ajouter au langage la possibilité de déclarer un attribut ou une méthode `private` ou `protected`. Un élément `private` ne doit alors être accessible que depuis le code de la classe courante. Un élément `protected` est accessible depuis le code de la classe courante ou de ses sous-classes (directes ou indirectes).

Déclarations en série

Ajouter la possibilité de déclarer simultanément plusieurs variables du même type, sous la forme

```
var int x, y, z;
```

Déclaration avec valeur initiale

Ajouter la possibilité, lors de la déclaration d'une variable ou d'un attribut, de lui fournir une valeur initiale, sous la forme

```
var int x = 1;
```

Dans le cas d'une variable globale (ou d'un attribut statique), cette valeur est initialisée une fois pour toutes au début de l'interprétation. Dans le cas d'une variable locale à une méthode, cette valeur est initialisée au début de chaque appel. Dans le cas d'un attribut, cette valeur est initialisée à chaque création d'une nouvelle instance.

Champs statiques

Ajouter au langage la possibilité de déclarer un attribut `static`. L'attribut n'est alors plus lié à chaque instance, mais à la classe elle-même, avec une valeur partagée pour toutes les instances.

Note : la combinaison de `static` et `final` impose que l'attribut soit initialisé lors de sa déclaration (extension **déclaration avec valeur initiale**), et pas dans le constructeur.

Test de type

Ajouter au langage un opérateur `instanceof` testant le type dynamique d'un objet. Plus précisément, `e instanceof t` vaut `true` si `e` a pour valeur un objet dont le type *dynamique* est un sous-type de `t`.

Note : pour l'essentiel, cet ajout concerne l'interprète. Cependant, dans certains cas il est possible dès le typage de prédire le résultat du test.

Transtypage

Ajouter au langage un opérateur de transtypage. L'expression `(t)e` est essentiellement équivalente à l'expression `e` du point de vue de l'interprète, mais est considérée par le typeur comme ayant le type `t`.

Plus précisément, on a les deux règles de typage suivantes :

```
E |- e : t'     t' <: t          E |- e : t'     t <: t'
-----------------------          -----------------------
     E |- (t)e : t                    E |- (t)e : t
```

et l'expression est rejetée au typage si le type statique `t'` de `e` et le type cible `t` ne sont pas sous-types l'un de l'autre.

En outre, on demande que l'interprète produise une erreur si le type dynamique de `e` n'est pas un sous-type de `t`, c'est-à-dire si la valeur de `e` n'est pas un objet d'une sous-classe de `t`.

Note : dans certains cas, ce dernier critère nécessite que l'interprète réalise un test du type dynamique de `e`. Dans d'autres cas, ce test est superflu (mais le faire ne pose pas de problème). Pour effectivement éviter de faire un test superflu, une possibilité est que le typeur "retire" l'opérateur de transtypage de l'AST lorsqu'il n'est pas nécessaire. Cette idée implique que le typeur ne se contente pas de vérifier qu'un programme est bien typé, mais également qu'il reconstruit une nouvelle version de l'AST qui peut être légèrement modifiée par rapport à la version issue de l'analyse syntaxique.

Super

Ajouter au langage la possibilité d'appeler une méthode `f(...)` avec la notation `super.f(...)`, pour appeler la version de la méthode `f` définie dans la classe mère.

Note : un tel appel doit être rejeté par le typeur lorsque cette version mère de la méthode `f` n'existe pas.

Tableaux

Ajoutez au langage la possibilité de manipuler des tableaux homogènes (c'est-à-dire, dont tous les éléments sont du même type). Vous avez besoin :

- d'un type pour les tableaux homogènes
- d'au moins une opération de création d'un tableau
- d'opérations d'accès en lecture et en écriture

Égalité structurelle

Ajouter au langage un opérateur `===` d'égalité structurelle et sa négation `=/=`. L'égalité structurelle répond aux critères suivants :

- chaque constante est égale à elle-même,
- deux objets sont égaux si et seulement si ils sont des instances de la même classe et ont pour chaque champ des valeurs structurellement égales,
- des valeurs de types différents ne sont pas comparables.

Classes et méthodes abstraites

Ajouter au langage la possibilité de déclarer des classes abstraites et des méthodes abstraites (mot-clé `abstract`). Une méthode abstraite a une signature mais pas de code. Une classe contenant une méthode abstraite doit elle-même être abstraite, y compris lorsque cette méthode est héritée. Une classes héritant d'une méthode abstraite peut en revanche donner une définition à cette méthode (qui n'est alors plus abstraite dans cette classe). On ne peut pas utiliser l'opérateur `new` avec une classe abstraite.

Surcharge statique

Ajouter au langage la possibilité de définir plusieurs méthodes de même nom, différenciées par les types attendus pour les paramètres. Lors d'un appel de méthode `e.f(e1, ..., eN)`, il faut choisir la bonne version de `f` en fonction des types *statiques* des paramètres effectifs `eK`.

Note : cette extension demande au typeur d'agir sur l'AST.

Attention : en présence de sous-typage, il faut choisir parmi les méthodes compatibles avec les types statiques des paramètres, celle qui a le type "le plus précis". On échoue au typage s'il y a ambiguïté.

Raffinements de l'analyse

Déclarations simplifiées

Assouplir le langage et adapter l'analyse syntaxique pour que les déclarations de variables, attributs et méthodes se fassent sans les mots-clés `var`, `attribute` et `method` (on obtient donc une syntaxe compatible avec java).

Note : si vous retirez ces mots-clés sans rien changer d'autre, cela crée des conflits.

"Missing semicolon"

Faire que votre analyseur produise des messages d'erreurs spécifiques pour certaines erreurs de syntaxe fréquentes.

"Did you mean 'recursion'?"

Lorsque le programme tente d'accéder à un identifiant inexistant, produire un message d'erreur proposant un identifiant ressemblant, s'il en existe.

Le processus ne peut pas aboutir en raison d'un problème technique

En cas d'erreur au moment du typage, faire en sorte que le message d'erreur contienne des informations de localisation de l'erreur. Notez que cela demande d'agir sur la définition de la syntaxe abstraite, pour adjoindre à chaque élément des informations de localisation, et sur l'analyse syntaxique pour effectivement collecter ces informations.

Syntaxe abstraite typée

Modifier la phase de typage pour qu'au lieu de simplement vérifier que le programme est bien typé, elle produise une nouvelle version de l'AST où chaque expression est annotée par son type (statique).

Comme l'extension précédente, cela nécessite de définir une variante de la syntaxe abstraite dans laquelle chaque expression peut être annotée.

Voici une manière de définit en caml une syntaxe abstraite des expressions où chaque sous-expression porte une annotation d'un certain type `t`.

```
type expr = { annot: t; expr: expr_ }
and expr_ =
  | Int   of int
  | Bool  of bool
  | Binop of binop * expr * expr
  ...
```