Université Bordeaux **Licence Informatique 2023-2024**
4TIN603U **Projet de Compilation**

Modalités pour rendre le travail

Ce projet est à réaliser à 2 personnes et à remettre le 25 avril 2024 sur le site Moodle du cours. Le dépôt devra contenir (où "nom" est remplacé par le nom d'un ou de plusieurs membres du groupe):

- Un fichier `nom.pdf` qui ne contiendra aucun code mais qui précisera:
  - Qui sont les auteurs du projet en détaillant qui a fait quoi.
  - Les réponses aux questions de chaque partie du projet.
  - Pour chaque partie du projet, les difficultés rencontrées s'il y en a, et qui fera un bilan de ce qui est fonctionnel et ce qui ne l'est pas.
- Un fichier `nom.tar.gz` ou `nom.tgz` qui contiendra le projet (incluant la partie extension telle que décrite précédemment).

Attention, le projet initial vous étant fourni compilant, un rendu qui ne compilerait pas aura automatiquement une note inférieure à la moyenne. Si vous n'arrivez pas à implémenter une fonctionnalité, on préfèrera une version compilable mais non fonctionnelle, avec une explication dans le rapport qu'un code approchant mais ne compilant pas. Rien ne vous empêche de mettre en commentaire des propositions ne compilant pas.

Dans la suite, on présente d'abord la structure du code de ce qui vous est fourni, puis on introduit pas à pas les caractéristiques du langage Pixlang dans la partie où elles sont utiles – chaque partie correspondant à une partie de l'interpréteur. Le but de ce projet est de réaliser un interpréteur pour le langage **Pixlang**. **Pixlang** est un langage de programmation de dessin que nous avons inventé pour cette occasion. L'annexe 4 récapitule la syntaxe et la sémantique du langage (cette dernière étant déjà implémentée pour vous).

Vous aurez à réaliser (dans l'ordre):

- un lexeur
- un parseur
- un renommeur de variable (pour désambiguïser les portées)
- expliquer une partie de ce que fait le vérificateur de type fourni
- un simplificateur d'arbre
- détailler (et éventuellement implémenter) une extension au-delà de ces points (avant-dernière section du présent document).

1 Base de code et bureaucratie

Si vous travaillez chez vous, en plus des paquets nécessaires pour le cours, il faut installer la bibliothèque `graphics` d'OCaml. Cela s'effectue via `opam` avec la commande `opam install graphics`. Au CREMI, tout est installé correctement et vous n'aurez à faire que les manipulations habituelles.

Le programme fourni contient :

- Un programme principal qui met en lien les différents éléments du compilateur. Il est dans le fichier `bin/main.ml` et produit l'exécutable `interpreter.out`. Ce fichier devrait être laissé tel quel.
- Un programme de visualisation du parseur dans le fichier `bin/visualiser.ml` qui produit l'exécutable `visualiser.out`. Il ne doit pas être modifié.
- Une définition du langage **Pixlang** sous forme d'arbre de syntaxe abstraite. Cette définition peut être trouvée dans `language/ast.mli`. Elle n'est pas à modifier dans la partie principale du projet.
- Un interpréteur du langage **Pixlang** (dans `interpreter/interpreter.ml`). Il est fonctionnel pour la partie principale du projet. Ne le modifiez que pour les éventuelles extensions.
- Une bibliothèque `Util` dans le dossier `util` qui contient trois modules utilitaires pour votre interpréteur.
- Un ensemble de programmes exemples dans le dossier `programs`. Ce dossier comprend quelques exemples corrects (dans `good_examples`) et quelques exemples avec des erreurs de sémantique (dans `semantic_incorrect`).

Vous pouvez compiler le projet avec la commande `make`. Les exécutables fournis affichent un message d'aide si vous les lancez sans argument. Les fichiers d'explication de menhir sont également généré. Vous pouvez générer la documentation du code fourni avec `make doc`.

Une documentation vous est fournie pour les modules implémentés que vous avez à utiliser, et quelques commentaires d'aide sont placés dans les fichiers à implémenter.

La version initiale de `interpreter.out` affiche une fenêtre contenant ce qui est affiché par le programme interprété, qui se ferme à l'appui d'une touche du clavier.

1.1 Barème indicatif

La partie principale du projet, qui consiste en l'implémentation de tout ce qui est décrit à la section 2, ainsi que les réponses aux questions qui y sont posées sera notée sur environ 16 points. À l'intérieur de cette partie, le lexeur et le parseur seront notés sur au maximum 8 points (peut-être moins), le reste étant divisé entre les passes d'analyse sémantique.

La partie extension (décrite à la section 3) sera notée sur environ 4 points.

Ce barème n'est qu'indicatif et pourra être ajusté à la correction, mais ne devrait pas trop différer.

On évaluera plus favorablement un projet qui s'est concentré sur quelques-uns des points à traiter, mais les a correctement réalisé et expliqué, qu'un projet qui s'est dispersé et a tout mal fait. C'est d'ailleurs pour s'adapter à ces différence que le barème ci-dessus est indicatif.

2 Le langage Pixlang

Pixlang est un mini-langage impératif qui permet de manipuler des valeurs de base (entiers, réels et booléens), ainsi que des types structurés prédéfinis (coordonnées, couleurs, pixels et listes). Il permet également d'afficher des pixels dans une fenêtre graphique via une instruction primitive du langage.

Il ne dispose pas de mécanisme d'appel de fonction, ni de support de chaînes de caractères basique.

Dans l'annexe 4 nous donnons l'ensemble de la syntaxe et de la sémantique du langage de façon formelle. On décrit ici successivement les éléments du langages par rapport à la partie du projet à compléter, ainsi que les questions auxquelles vous devez répondre en lien avec cette partie.

2.1 Éléments lexicaux

Dans cette partie, on décrit les *éléments lexicaux* (ou tokens) de Pixlang. Cette partie sert à compléter le fichier `Lexer.mll`.

Mots clé Les mots-clés du langages sont soit des séquences de lettres commençant par une majuscule, soit des séquences de caractères spéciaux. Voici leur liste exhaustive:

`And`, `Blue`, `Bool`, `Color`, `Coord`, `Cos`, `Draw`, `Else`, `False`, `Floor`, `For`, `Foreach`, `From`, `Green`, `Head`, `If`, `In`, `Int`, `List`, `Not`, `Or`, `Pixel`, `Print`, `Real`, `Real_of_int`, `Red`, `Set`, `Sin`, `Step`, `Tail`, `To`, `True`, `X`, `Y`, `Pi`.

`$< >$ + - * / % = <> <= >= < > : :: . ( ) [ ] , ;`

Dans la définition de la syntaxe du langage (section suivante), on les utilisera tels quels.

Identificateurs Les identificateurs commencent par une lettre minuscule non accentuée éventuellement suivie d'une série de lettres alphanumériques et d'underscore. Exemples d'identificateurs: `x`, `i`, `premier_nombre`, `value12`

Dans la définition de la syntaxe du langage, on utilisera `{id}` pour désigner un tel identificateur.

Entiers Les entiers peuvent être fournis sous deux formats :

- En décimal (une suite non-vide de chiffres entre 0 et 9).
- En hexadécimal ("0x" suivi d'une suite de chiffres entre 0 et 9, ou entre A et F).

Dans la définition de la syntaxe du langage, on utilisera `{int}` pour désigner l'entier obtenu.

Indication: la fonction OCaml `int_of_string` accepte ces deux formats.

Réels Les nombres réels peuvent être fournis en notation décimale avec virgule, c'est-à-dire, une suite de chiffres entre 0 et 9 (potentiellement vide), suivie d'un '.', suivi d'une suite de chiffres entre 0 et 9 (potentiellement vide).

Dans la définition de la syntaxe du langage, on utilisera `{real}` pour désigner le réel obtenu.

Commentaires Les commentaires du langage sont sous deux formats :

- N'importe quelle séquence de caractères commençant par "//" et terminant par un retour à la ligne.
- N'importe quelle séquence de caractères commençant par "/*" et terminant à la première occurrence de "*/".

Ils ne correspondent à aucun éléments lexicaux du langage et doivent donc être ignorés.

On prêtera cependant à compter correctement le nombre de lignes dans les commentaires qui contiennent des retours à la ligne.

Autres Les caractères d'espacement ' ', '\r' et '\t' sont ignorés, c'est-à-dire acceptés mais ne correspondent à aucun élément lexical.

Les retours à la ligne '\n' sont de même ignorés mais doivent incrémenter le compteurs de lignes du buffer.

Le caractère de fin de fichier 'eof' correspond à un élément lexical spécial qu'on notera `EOF`.

Questions à faire apparaître dans le document Dans votre document, vous préciserez les difficultés rencontrées si votre lexeur n'accepte pas tout ce qui est décrit ici.

Vous expliquerez la solution mise en œuvre pour reconnaître les commentaires sur plusieurs lignes tout en maintenant le compteur de ligne du buffer cohérent avec le fichier analysé.

2.2 Grammaire du langage

Dans cette partie, on définit la *grammaire* du langage, c'est-à-dire qu'on décrit quelle séquence de tokens (définis ci-avant) correspond à quel arbre de syntaxe abstraite. Les cas possibles pour les AST sont définis dans le fichier `langage/ast.mli`.

Cette partie sert à compléter le fichier `Parser.mly`. Évidemment, si ce fichier ressemblera fortement à ce qui est décrit ici, il peut être nécessaire qu'il diffère (priorités, non-terminaux techniques et/ou inliné, etc).

Les différents types de nœuds d'un AST (associés aux non-terminaux de la grammaire à leur droite) sont les suivants (correspondant aux différents types de `Ast.mli`) :

- `type_expression`, associé à `#type#` dans la suite.
- `binary_operator`, associé à `#binop#` dans la suite.
- `unary_operator`, associé à `#unop#` dans la suite.
- `field_accessor`, associé à `#field#` dans la suite.
- `expression`, associé à `#expr#` dans la suite.
- `statement`, associé à `#stmt#` dans la suite.
- `argument`,associé à `#arg#` dans la suite.
- `program`, associé à `#prg#` dans la suite.

Un *programme* du langage **Pixlang** est un AST dont la racine est de type `program`.

La plupart des nœuds de l'AST disposent d'une *annotation* (cf. `Ast.mli`). Cette annotation ne correspond pas à un élément syntaxique, et dans la description suivante sera figurée par un `_`. Cette annotation servira à stocker les informations de la partie du fichier l'ayant engendrée (à initialiser dans le parseur) et également à stocker le type lors de l'analyse de types.

Dans la suite, on décrit pour chacun des types définis plus haut les séquences de tokens correspondant à leurs constructeurs. On donne ces définitions sous la forme d'un tableau.

Dans ce tableau, lorsque plusieurs terminaux ou non-terminaux apparaissent, il seront annotés par un numéros, par exemple `#arg1#` dénote le premier non-terminal de type `argument` de la production. Dans la définition de l'AST, on utilisera le nom d'un non-terminal ou d'un terminal pour désigner l'AST correspondant à cet élément.

Enfin, on dénotera par `...` la présence d'une répétition dans la règle. Par exemple `#arg1# ; ... ; #argk#` dénotera une liste de non-terminaux `#arg#` de taille k, séparés par des point-virgules.

Sauf mention explicite, les listes pourront être vide.

En cas de doute, vous pouvez vous reporter aux exemples du dossier `programs` qui contient des programmes définis par cette grammaire (et qu'il vous faudra analyser).

Program Un programme dispose d'une liste d'arguments entourée de `<` et `>`, séparés de point-virgules (qui peut être absente), suivie d'un statement. Le dernier argument n'est pas suivi d'un point-virgule.

`#prg#`	AST
`< #arg1# ; ... ; #argk# > #stmt#`	`Program([#arg1#;...;#argk#],#stmt#)`
`#stmt#`	`Program([],#stmt#)`

Argument Un argument est un type suivi du nom de l'argument, les deux étant séparés par `:`.

`#arg#`	AST
`#type# : {id}`	`Argument({id},#type#,_)`

Statement Un statement consiste l'une des instructions possibles du langage.

Attention, ici, les statement ne sont pas nécessairement terminés par un `;`, ils servent par contre de séparateur à l'intérieur d'un bloc.

Le `Else` est facultatif dans le if-then-else. Un `Else` se rattache toujours au `If` non-terminé le plus proche.

#stmt#	AST
Set (#expr1# , #expr2#)	Affectation(#expr1#,#expr2#,_)
#type# : {id}	Declaration({id},#type#,_)
$< #stmt1#; ... ; #stmtk# >$	Block([#stmt1#;...;#stmtk#],_)
If (#expr#) #stmt1# Else #stmt2#	IfThenElse(#expr#,#stmt1#,#stmt2#,_)
If (#expr#) #stmt#	IfThenElse(#expr#,#stmt#,Nop,_)
For {id} From #expr1# To #expr2# Step #expr3# #stmt#	For({id},#expr1#,#expr2#, #expr3#,#stmt#,_)
Foreach {id} In #expr# #stmt#	Foreach({id},#expr#,#stmt#,_)
Draw (#expr#)	Draw_pixel(#expr#,_)
Print (#expr#)	Print(#expr#,_)
	Nop

Expression Une expression représente une expression calculant une valeur du langage. Ce sont soit des constantes, soit des expressions composées avec des opérateurs.

La notation `::` est prioritaire sur `.` qui est elle-même prioritaire sur les opérateurs binaires. Ces deux notations sont associatives à droite.

Les opérateurs unaires sont prioritaires sur toutes les notation précédentes.

#expr#	AST
{int}	Const_int ({int},_)
{real}	Const_real({real},_)
Pi	Const_real(Float.pi,_)
True	Const_bool (true,_)
False	Cont_bool (false,_)
{id}	Variable ({id},_)
Coord(#expr1#,#expr2#)	Coord(#expr1#,#expr2#,_)
Color(#expr1#,#expr2#,#expr3#)	Color(#expr1#,#expr2#,#expr3#,_)
Pixel(#expr1#,#expr2#)	Pixel(#expr1#,#expr2#,_)
#expr1# #binop# #expr2#	Binary_operator(#binop#,#expr1#,#expr2#,_)
#unop# #expr#	Unary_operator(#unop#,#expr#,_)
#expr# . #field#	Field_accessor(#field#,#expr#,_)
[#expr1# , ... , #exprk#]	List([#expr1# ; ... ; #exprk#],_)
#expr1# :: #expr2#	Append(#expr1#,#expr2#,_)
(#expr#)	#expr#

Opérateurs binaires Les opérateurs binaires disposent d'une priorité et d'une associativité, qui sont précisés dans le tableau. Les priorité ne sont pas à interpréter comme une valeur exacte, mais plutôt comme un ordre relatif. Les nombres les plus grands représentent la plus forte priorité.

#binop#	AST	priorité	associativité
+	Plus	5	gauche
-	Minus	5	gauche
*	Times	6	gauche
/	Div	6	gauche
%	Rem	6	gauche
And	And	3	gauche
Or	Or	2	gauche
=	Equal	4	non-associatif
<>	Diff	4	non-associatif
<	Lt	4	non-associatif
>	Gt	4	non-associatif
<=	Leq	4	non-associatif
>=	Geq	4	non-associatif

Opérateurs unaires Les opérateurs unaires sont les suivants :

#unop#	AST
-	Opposite
Not	Not
Head	Head
Tail	Tail
Floor	Floor
Real_of_int	Real_of_int
Cos	Cos
Sin	Sin

Field Les champs sont les suivants :

#field#	AST
Color	Color_field
Coord	Coord_field
X	X_field
Y	Y_field
Red	Red_field
Green	Green_field
Blue	Blue_field

Types Les types sont les suivants :

#type#	AST
Int	Type_int
Real	Type_real
Bool	Type_bool
Coord	Type_coord
Color	Type_color
Pixel	Type_pixel
List (#type#)	Type_list(#type#)

Dans l'AST apparaît un type supplémentaire, `Type_generic`. Ce type n'est pas un type syntaxique du langage. Il ne sera utilisé là que lors de l'analyse de type, pour gérer les listes vides.

Questions à faire apparaître dans le document Dans votre document, vous préciserez les difficultés rencontrées si votre parseur n'accepte pas tout ce qui est décrit ici.

Vous pouvez bien évidemment jouer avec menhir pour répondre à ces questions.

1. On considère la séquence suivante : `If #expr# If #expr# #stmt# Else #stmt#`

   On considère pour cette question que `#expr#` et `#stmt#` sont des terminaux (i.e., on ne cherchera pas à les «étendre»).

(a) Donnez les deux arbres de dérivations possible de cette séquence dans la grammaire décrite plus haut.

(b) Donnez l'état de l'automate LR0 où apparaît le conflit qui montre l'existence de ces deux arbres.

(c) Quelle annotation permet d'obtenir l'arbre cohérent avec la priorité décrite dans ce document ?

(d) Pouvez-vous via une annotation obtenir le comportement inverse ? Pourquoi ?

2. Choisissez un conflit shift-reduce possible dans votre grammaire sans annotation (qui n'est pas celui de la question précédente), et expliquez quelles annotations de priorité vous avez mis pour le résoudre, et son effet sur les arbres acceptés (quels arbres sont privilégiés, lesquels sont ignorés). Vous illustrerez un exemple où ce conflit pourrait arriver et les deux arbres mis en jeu via une séquence de tokens.

2.3 Non-redondance des noms de variables

Cette partie consiste à implémenter une passe de transformation de l'AST qui va renommer les variables qui seraient redondantes de manière à ce qu'une variable existante dans la portée courante ne soit pas redéfinie (on va simplement changer son nom). Cette passe a pour but de simplifier l'analyse de type qui suivra.

Cette partie sert à implémenter `analyser/renaming.ml`, et les fonctions de renommage qu'elle contient.

Ces fonctions vont toutes disposer, outre d'un argument étant le nœud d'un AST, d'un argument de type `int Environement.t`. Cet argument contiendra, pour chaque variable, son nombre de redéfinition dans la portée courante. Elles renvoient un AST qui est l'AST d'entrée modifié (les variables sont renommées).

L'idée du renommage est simplement d'incrémenter l'entier associé à une variable à chacune de ses déclarations, et à chaque fois que la variable est utilisée (y compris la déclaration donc), de remplacer le nom de la variable `x` par `x#i` où i est l'entier associé à la variable dans l'environnement. Si i est égal à 0, on ne renomme pas la variable (cela correspond à la première occurence du nom, qu'on peut donc se passer de renommer).

Il convient évidemment de copier l'environnement pour les nœuds modifiant la portée des déclaration de manière à ne pas faire de renommage intempestif.

Le reste de cette section détaille l'algorithme pour chaque cas de l'AST où l'action à faire n'est pas triviale (i.e., se rappeler récursivement sur tous les sous-nœuds avec les mêmes arguments).

On considère `env` un environnement de type `int Environmenent.t`. Les valeurs non-utilisées sont notées par `_` (elles sont alors inchangées). Si on ne met rien dans la case ≪Nouvel AST≫, c'est qu'aucune modification particulière n'est faite sur ce nœud (outre les modifications de ses enfants).

Arguments

AST	Condition sur `env`	Action dans `env`	nouvel AST
`Argument(id,_,_)`	`id` non-def	associer 0 à `id`	`Argument(id,_,_)`
`Argument(id,_,_)`	`id` associé à i	associer $j = i + 1$ à `id`	`Argument(id#j,_,_)`

Statements

AST	Condition et action sur `env`	nouvel AST
`Declaration(id,_,_)`	si `id` non-def, associer 0 à `id`	`Declaration(id,_,_)`
`Declaration(id,_,_)`	si `id` associé à i, associer $j = i + 1$ à `id`	`Declaration(id#j,_,_)`
`Block(l,_)`	Utiliser une copie de `env` dans l'analyse de `l`	
`IfThenElse(test,th,el,_)`	Utiliser des copies de `env` dans les analyses de `th` et de `el`	
`For(id,e1,e2,e3,body,_)`	Utiliser une copie de `env` dans l'analyse de `body` où `id` est associé à 0 si non-déf et à $i + 1$ s'il est associé à i dans `env`	
`Foreach(id,test,body,_)`	Utiliser une copie de `env` dans l'analyse de `body` où `id` est associé à 0 si non-déf et à $i + 1$ s'il est associé à i dans `env`	

Expression

AST	Condition et action sur `env`	Nouvel AST
`Variable (id,_)`	`id` associé à 0 ou non-défini	`Variable (id,_)`
`Variable (id,_)`	`id` associé à $j > 0$	`Variable (id#j,_)`

Questions à faire apparaître dans le document Dans votre document, vous préciserez les difficultés rencontrées si votre renommage n'effectue pas tout ce qui est décrit ici.

1. Pourquoi n'est-il pas gênant que dans deux blocs disjoints (pas l'un dans l'autre) un même nom soit utilisé pour des variables locales à ces blocs ?

2. Dans le programme suivant (il s'agit du programme `renaming.pix`, qui n'a pas d'intérêt particulier autre que pour cet exercice), indiquez comment le renommage des variables sera effectué :

```
<
    Int : x;
    Real : y
>
$<
```

```
    Set(x,2*x);
    Real : x;
    Set(x,2.0*y);
    Int : y;
    Set(y,Floor(x));
    $<
        Int : x;
        Set(x,2*y);
        Int : y;
        Set(y,2*x);
    >$;
    Set(y,2*y);
    $<
        Coord : x;
        Set(x,Coord(y,y));
        Color : y;
        Set(y,Color(x.X,x.X,x.X));
        Draw(Pixel(x,y));
    >$;
    Set(x,2.2*x);
>$
```

2.4 Typage

Cette partie se concentre sur le vérificateur de type qui est implémenté dans le fichier `analyser/type_analyser.ml`. Le code vous est fourni, vous avez juste à répondre aux questions à la fin de cette section (et du coup à comprendre ce que fait le code).

Les fonctions qui traitent les statements et les expressions disposent, outre de leur argument étant le nœud de l'AST, d'un argument de type `Ast.type_expression Environment.t`. Cet argument contient pour chaque variable le type qui lui a été déclaré.

Ces fonctions disposent également d'un argument de type `Error_report.t` qui permet de reporter des erreurs à l'utilisateur. On le remplira lorsqu'on rencontrera un cas où le typage est erroné.

Ces fonctions travaillent uniquement par effet de bord (pas de type de retour), sauf le typage des expressions, qui renvoie le type de l'expression (pour plus de facilité d'usage).

L'idée générale de l'algorithme est le suivant : chaque déclaration de variable va modifier l'environnement des types, en associant le type fourni à la variable correspondante. Les boucles `For` et `Foreach` associent également le type de la variable de boucle (le type des bornes pour un `For` (attention, si ces trois types sont différents, une erreur est reportée), le type contenu dans la liste pour un `Foreach`). On ne type que les expressions. Si on a une constante, le type est porté par le constructeur. Si on a une variable, le type est celui associé à la variable dans l'environnement (sinon, on notifie une erreur). Dans les autres cas, on commence par typer les sous-expressions, et on récupère leur type dans leurs annotations. Ensuite, en fonction du cas, on détermine si les types des sous-expressions sont compatibles, et si oui, on détermine le type de l'expression courante, qu'on stocke alors dans l'annotation associée au nœud de l'arbre. Si non, on notifie une erreur à l'utilisateur.

Dans la suite, on précise pour les expressions et les opérateurs leur comportement par rapport au type.

Expressions

AST	Type
`Const_int (_,_)`	`Type_int`
`Const_real(_,_)`	`Type_real`
`Const_bool (_,_)`	`Type_bool`
`Variable (id,_)`	le type associé à `id` dans `env`
`Coord(x,y,_)`	`Type_coord`. `x` et `y` doivent être des `Type_int`
`Color(r,g,b,_)`	`Type_color`. `r`, `g` et `b` doivent être des `Type_int`
`Pixel(p,c,_)`	`Type_pixel`. `p` doit être `Type_coord` et `c`, `Type_color`
`Binary_operator(op,e1,e2,_)`	`e1` et `e2` doivent être compatibles avec `op`, le type est fixé plus bas
`Unary_operator(op,e,_)`	`e` doit être compatible avec `op`, et le type est fixé plus bas.
`Field_accessor(field,e,_)`	`field` doit être compatible avec le type de `e` on a alors le type associé à `field`
`List([e1 ; ... ; ek],_)`	tous les `ei` doivent être du même type `t`. Le type est `Type_list(t)` La liste vide est de type `Type_list(Type_generic)`
`Append(e,l,_)`	`Type_list(t)` si `e` est de type `t` et `l` de type `Type_list(t)` ou `Type_list(Type_generic)`

Opérateurs binaires On présente maintenant les types associés aux opérateurs binaires. Pour des raisons de place, on va les regrouper par comportements : Les opérateurs booléens, `And` et `Or` doivent avoir leurs deux opérandes de type `Type_bool` et donnent un résultat de même type.

Les opérateurs de comparaisons doivent avoir leurs deux opérandes de même type (tous types acceptés pour `Eq` et `Diff`, seuls les entiers, réels et booléens pour les autres) et donnent un résultat de type `Type_bool`.

Les opérateurs arithmétiques `Plus`, `Minus`, `Times`, `Div` et `Rem` sont définis pour des opérandes de même type parmi `Type_int`, `Type_real`, `Type_coord`, `Type_color` et `Type_pixel`. Le résultat est de même type. Par ailleurs, si l'un des opérande est de type `Type_int` et l'autre de type `Type_coord` ou `Type_color`, il est également défini, et le résultat est de type `Type_coord` ou `Type_color` (le même que l'opérande).

Enfin, l'opérateur `Plus` est également défini sur deux opérandes de type `Type_list(t)`

(pour tout type `t`), et le résultat est de même type. Si l'une des deux liste est de type `Type_list(Type_generic)`, il n'y a pas d'erreur et c'est l'autre type qui l'emporte.

Par ailleurs, pour les opérateurs arithmétiques et de comparaisons, si l'un des opérande est de type `Type_int` et l'autre de type `Type_real`, on ne notifie pas une erreur, mais uniquement un avertissement à la place. Dans le cas des opérateurs arithmétiques, le résultat sera alors `Type_real`.

Opérateurs unaires Les types de opérateurs unaires sont les suivants:

AST	Type compatible	Type de sortie
`Opposite`	`Type_int` ou `Type_real`	Le même
`Not`	`Type_bool`	`Type_bool`
`Head`	`Type_list(t)`	`t`
`Tail`	`Type_list(t)`	`Type_list(t)`
`Floor`	`Type_real`	`Type_int`
`Real_of_int`	`Type_int`	`Type_float`
`Cos`	`Type_float`	`Type_float`
`Sin`	`Type_float`	`Type_float`

Questions à faire apparaître dans le document

1. Pourquoi a-t'on besoin de `Type_generic` pour la liste vide ?
2. Pourquoi doit-on réaliser des copies des environnements avant de vérifier la cohérence des types à l'intérieur des blocs ?
3. Donnez un tableau similaire à celui des opérateurs unaires qui précise le typage des champs (fields).
4. Expliquez quelles sont les deux erreurs qu'on peut détecter lorsqu'on traite un statement `For(x,start,last,step,body)` lors de l'analyse de type.

2.5 Simplification de terme

Cette partie consiste à simplifier l'AST en simplifiant les expressions constantes et en retirant les instructions sans effet.

Cette partie sert a implémenter le fichier `analyser/simplifier.ml`.

Les fonctions de ce fichier prendront en entrée un AST (pour chaque type de nœud) et renverront un AST de même type.

L'idée de cette partie consiste à générer un nouvel AST en parcourant l'AST reçu en paramètre en simplifiant les cas suivant (après simplification des sous-arbres) :

- Si une opération unaire ou binaire a pour sous-arbres uniquement des constantes, on effectue le calcule et on renvoie la constante correspondante.
- On a le même comportement pour les champs ou `Append`.
- De plus, pour les opération binaires arithmétiques, si on a une opération entre une constante entière et un nœud de type `Coord` ou `Color`, on applique l'opération entre l'entier et chacun des composants de la coordonnée ou de la couleur.
- Si le test d'un `If` est une constante, on remplace le `If` par la branche qui sera systématiquement prise.
- Si un `For` a une borne de départ et une borne d'arrivée qui sont des constantes telles que la borne de départ est strictement supérieure à celle d'arrivée, on remplace la boucle par un bloc vide.
- Si un `Foreach` a pour argument une liste vide, on le remplace par un bloc vide.
- Si on a un arbre de la forme `Floor(Real_of_int(n,a1),a2)`, on garde uniquement le nœud `n`.

Dans tous ces cas, il conviendra de placer sur le nœud obtenu après simplification, l'annotation que contenait le nœud avant simplification (pour conserver des erreurs cohérentes).

Par ailleurs, on va dans ce même algorithme, rajouter les cast implicites qui auront été déterminés lors du typage (cf, section précédente). Si un nœud `Binary_operator` a un sous-arbre de type `Type_real`, et l'autre de type `Type_int`, celui de type `Type_int` se voit remplacer par un nœud `Unop(Real_of_int,n,a)` où `n` est le sous-arbre d'origine (simplifié) et `a` une annotation vierge (sans position), mais avec le bon type. Il vous faudra pour cela, une fonction récupérant l'annotation d'une expression (ce qui n'est pas très dur).

Questions à faire apparaître dans le document Dans votre document, vous préciserez les difficultés rencontrées si votre simplificateur n'effectue pas tout ce qui est décrit ici.

1. Pourquoi peut-on éliminer les `For` dans le cas décrit ci-dessus ?
2. Pourquoi ne simplifie t'on pas les nœuds de la forme `Real_of_int(Floor(n,a1),a2)` ?

3 Extension

En plus de la base commune ci-dessus, qui vous donnera un interpréteur fonctionnel du langage, nous vous demandons de choisir une tâche en plus à, expliquer dans le document à rendre. Deux des extensions ne vous demandent que de détailler ce qui serait à mettre en œuvre (dans le style du présent sujet), et deux étant beaucoup plus proche que ce qui est décrit, vous demandent d'implémenter au moins partiellement ces idées (bien évidemment, celles où on ne vous demandent pas l'implémentation peuvent tout de même l'être, et si c'est bien fait, cela sera apprécié, mais on pourra obtenir la note maximale sans ces implémentations).

Ces extensions sont volontairement moins détaillées que les tâches précédente : l'un de vos objectifs est justement d'expliciter et de justifier la tâche supplémentaire que vous choisissez.

Traitez l'une des sous-sections suivantes :

3.1 Simplification++

Cette tâche consiste à simplifier encore plus de cas que ceux décrit dans la simplification (qui est assez basique).

Vous détaillerez quelques cas de simplification supplémentaires que vous pouvez ajouter au simplificateur de termes, en justifiant soigneusement pourquoi vous pouvez simplifier ce cas.

L'un de ces cas devra concerner les boucles `For`, dans le cas où elle ne s'exécutent qu'une fois.

Un autre de ces cas devra concerner les termes arithmétiques, en tentant de minimiser le nombre d'opérations exécutées en utilisant des propriétés mathématiques, et en supposant que la multiplication, la division et le modulo sont plus cher que l'addition, la soustraction, et la multiplication/division par une puissance de deux (ce qui est vrai sur un processeur, mais faux dans ce projet).

Détaillez d'autres cas que vous pensez simplifiables.

Vous rajouterez quelques programmes sur lesquels vos simplifications ont un effet (et les mentionnerez clairement dans votre document).

Si vous choisissez cette partie, on attendra qu'au moins le cas du `For` soit implémenté et expliquée, et que la simplification des termes arithmétiques soit bien expliquée (à défaut d'être implémentée) pour attribuer une note correcte à cette partie.

3.2 Propagation de constante

Dans certains programmes, des variables contiennent en fait une constante à une ligne du programme quelque soit l'exécution (et son moment) qu'on est capable de déterminer à la compilation. Il est dans ce cas plus économe dans le programme produit de remplacer l'utilisation de ces variables directement par les constantes correspondantes.

L'idée est de, pour chaque variable, déterminer pour chaque utilisation de la variable, si elle ne peut contenir qu'une seule valeur (connue) ou si plusieurs sont possible. La tâche est relativement aisée pour les programmes sans boucles, mais demande de réfléchir un peu pour les boucles (et les if-then-else). Attention, pour cette tâche, il faudra traiter soigneusement les blocs (et les structures de contrôle) : on ne pourra pas se contenter de les traiter avec une copie de l'environnement qu'on oublie ensuite. Il faudra prendre en compte l'effet du bloc.

Vous décrirez en détail (i.e., dans le style des parties précédentes) comment la réaliser.

Si vous implémentez cette tâche, vous la placerez dans un fichier `analyser/constant_prop.ml`, et vous ajouterez l'appel de sa fonction principale dans `analyser/analyser.ml`, avant l'appel au simplificateur. Vous aurez pour cela besoin d'un type représentant le statut de chaque variable, qui devrait être:

```
type is_constant = Uninitialised | Constant of int | Not_constant
```

Vous ajouterez également quelques exemples de programmes (et les mentionnerez clairement dans votre document) où votre analyse a un effet.

Pour avoir une note correcte sur cette partie, seule la description en détail sera attendue (l'implémentation n'étant pas si facile que ça), mais si vous l'implémentez, vous aurez

évidemment plus de points. Vous expliquerez également pourquoi il est mieux de réaliser cette passe avant la simplification.

3.3 Initialisation et utilité des variables

On peut ajouter une passe qui détermine si les variables sont initialisées avant utilisation, et si les variables sont utiles.

Vous décrirez (dans le style des sections précédentes) et implémenterez une passe supplémentaire à l'analyseur qui :

- si une variable est utilisée sans avoir été initialisée, reporte un avertissement à l'utilisateur.
- si une variable est déclarée, mais n'est jamais utilisée, reporte un avertissement à l'utilisateur (même si elle est initialisée).

Attention, pour cette tâche, il faudra traiter soigneusement les blocs (et les structures de contrôle) : on ne pourra pas se contenter de les traiter avec une copie de l'environnement qu'on oublie ensuite. Il faudra prendre en compte l'effet du bloc. C'est là la difficulté principale de cette tâche.

Vous placerez cette tâche dans un fichier `analyser/initialiser.ml`, et l'appellerez dans `analyser/analyser.ml` après la phase de renommage des variables.

Vous expliquerez en outre pourquoi on place cette tâche après la phase de renommage.

Vous ajouterez également quelques programmes (et les mentionnerez clairement dans votre document) où votre analyse détecte des erreurs d'initialisation ou de déclaration inutiles qui seraient autrement restées silencieuses.

Pour avoir une note correcte sur cette partie, on attendra au minimum une explication claire de l'algorithme et une implémentation partielle (mais si elle est fonctionnelle, ce sera évidemment mieux).

3.4 Inférence de type

Sur un langage avec un typage aussi simple (et sans fonction), il n'est pas si difficile de déterminer automatiquement les types au lieu de simplement en vérifier la cohérence.

On peut rajouter syntaxiquement un mot-clé (par exemple `Auto`) comme nom de type, correspondant à `Type_generic`, et modifier l'analyse de type de manière à déterminer le type des variables ainsi déclarées (et évidemment ensuite à modifier l'AST pour avoir les bonnes déclarations de type).

Vous décrirez en détail (i.e., dans le style des sections précédentes) ce qu'il faut faire pour y parvenir.

Seule cette explication détaillée sera exigée de vous sur cette partie.

Attention, pour cette tâche, il faudra traiter soigneusement les blocs (et les structures de contrôle) : on ne pourra pas se contenter de les traiter avec une copie de l'environnement qu'on oublie ensuite. Il faudra prendre en compte l'effet du bloc.

Si toutefois vous souhaitiez l'implémenter, faites-le dans un nouveau fichier du dossier `analyser` (par exemple, `type_inferer.ml`) et remplacer l'appel à `type_analyser` par votre nouvelle fonction. Il vous faudra également rajouter un cas dans le parseur pour pouvoir réaliser cette tâche (mais ce devrait être élémentaire). Dans ce cas, vous rajouterez également quelques programmes (et les mentionnerez clairement dans votre document) au dossier programmes qui permettent d'illustrer votre implémentation.

3.5 Enrichissement du langage

Vous pouvez rajouter du sucre syntaxique au langage, ainsi qu'une boucle.

Rajoutez une boucle while standard au langage (à l'AST et au parseur), et modifier toutes vos passes pour en tenir compte (y compris l'interpréteur). Vous expliquerez clairement comment vous avez traité le cas de cette boucle dans l'interpréteur et dans les différentes passes (dans le style des sections précédentes).

Vous rajouterez également au moins deux constructions de sucre syntaxique dans le lexeur et le parseur, c'est-à-dire des constructions syntaxiques qui ne correspondent pas à un vrai nœud de l'AST, mais à plusieurs existant déjà, que vous reconstruirez directement dans le parseur. Vous expliciterez dans le document la syntaxe, et l'arbre correspondant.

Vous pouvez par exemple rajouter un opérateur ternaire (comme en C), des opérations mathématiques simples (carré, etc), ou des normaliseurs de couleurs (par exemple, un opérateur qui remplace une couleur (r,g,b) par (g,b,r) ou tout autre opération rigolote).

Vous illustrerez ces constructions par quelques exemples de programmes (et les mentionnerez clairement dans votre document).

4 Annexe : Syntaxe et sémantique de Pixlang

Les types de données `value` de la mémoire sont les mêmes que ceux du langage. On considère que chacun occupe la même taille en mémoire (simplification).

Le modèle mémoire de Pixlang est un tableau indexé par des variables qui sont des `string`. Un état de la mémoire est une fonction $\rho : \texttt{string} \to \texttt{value}$.

Sur un état de mémoire, on peut faire une affectation : $\rho[x \leftarrow v]$ est l'environnement égal à ρ mais dans lequel la valeur mémoire de x est v.

Les opérations applicables sur les `value` sont les opérations mathématiques usuelles (sur les entiers et les flottants), ainsi que la concaténation des listes.

On donne dans ce qui suit un tableau récapitulant la syntaxe texte, la syntaxe AST correspondante et la valeur des expressions dans un environnement ρ. Les annotations n'étant pas utiles dans l'interpréteur, on les figure ici par `_`.

AST	valeur : $[e]_\rho$
`Const_int (i,_)`	l'entier i
`Const_real(f,_)`	le flottant f
`Const_bool (b,_)`	le booléen b
`Variable (name,_)`	$\rho(name)$
`Coord(e1,e2,_)`	$([e1]_\rho, [e2]_\rho)$ si cela forme un couple d'entiers, indéfini sinon
`Color(e1,e2,e3,_)`	$([e1]_\rho, [e2]_\rho, [e3]_\rho)$, si cela forme un triplet d'entier, indéfini sinon
`Pixel(e1,e2,_)`	$([e1]_\rho, [e2]_\rho)$, si $[e1]_\rho$ est une coordonnée et $[e2]_\rho$ est une couleur
`Binary_operator(op,e1,e2,_)`	$[\![op]\!][e1]_\rho[e2]_\rho$ $[\![op]\!]$ est défini plus bas
`Unary_operator(op,e,_)`	$[\![op]\!][e]_\rho$ $[\![op]\!]$ est défini plus bas
`Field_accessor(field,e,_)`	$[\![field]\!][e]_\rho$ $[\![field]\!]$ est défini plus bas
`List([e1;...;ek],_)`	la liste $[[e1]_\rho; \cdots ; [ek]_\rho]$ toutes les expressions doivent être de même type
`Append(e1,e2,_)`	la liste $[e1]_\rho :: [e2]_\rho$ $e1$ doit être d'un type `t` et $e2$ de type `Type_list(t)`

On donne maintenant la syntaxe et la sémantique des opérations binaires. Elle est donnée de manière précise, mais informelle :

AST	$[\![op]\!]$
`Plus`	sur deux entiers ou deux flottants : leur addition sur deux positions, deux couleurs, ou deux points : leur addition terme à terme sur deux listes : leur concaténation sur un entier et une coordonnée/couleur : addition de l'entier terme à terme
`Minus`	sur deux entiers ou deux flottants : leur soustraction sur deux positions, deux couleurs, ou deux points : leur soustraction terme à terme sur un entier et une coordonnée/couleur : soustraction de/par l'entier terme à terme
`Times`	sur deux entiers ou deux flottants : leur multiplication sur deux positions, deux couleurs, ou deux points : leur multiplication terme à terme sur un entier et une coordonnée/couleur : multiplication de l'entier terme à terme
`Div`	sur deux entiers ou deux flottants : leur division sur deux positions, deux couleurs, ou deux points : leur division terme à terme sur un entier et une coordonnée/couleur : division de/par l'entier terme à terme
`Rem`	sur deux entiers ou deux flottants : leur modulo sur deux positions, deux couleurs, ou deux points : leur modulo terme à terme sur un entier et une coordonnée/couleur : module de/par l'entier terme à terme
`And`	Et booléen
`Or`	Ou booléen
`Equal`	égalité sur tous les types
`Diff`	inégalité sur tous les types
`Lt`	strictement plus petit sur entiers et flottants
`Gt`	strictement plus grand sur entiers et flottants
`Leq`	inférieur ou égal sur entiers et flottants
`Geq`	supérieur ou égal sur entiers et flottants

On donne maintenant la syntaxe et la sémantique des opérations unaires, dans le même style que celle des opérations binaires :

AST	$[\![op]\!]$
`Opposite`	opposée d'un entier ou d'un flottant.
`Not`	négation binaire
`Head`	Renvoie le premier élément d'une liste
`Tail`	Renvoie une liste sans son premier élément
`Floor`	Renvoie la partie entière d'un flottant
`Float_of_int`	Renvoie le flottant égal à l'entier sur lequel il s'applique
`Cos`	Cosinus d'un flottant
`Sin`	Sinus d'un flottant

La sémantique des champs est la suivante :

AST (champs)	$[\![field]\!][e]_\rho$
`Color_field`	c si $[e]_\rho$ est le pixel (p, c)
`Position_field`	p si $[e]_\rho$ est le pixel (p, c)
`X_field`	x si $[e]_\rho$ est la coordonnée (x, y)
`Y_field`	y si $[e]_\rho$ est la coordonnée (x, y)
`Red_field`	r si $[e]_\rho$ est la couleur (r, g, b)
`Green_field`	g si $[e]_\rho$ est la couleur (r, g, b)
`Blue_field`	b si $[e]_\rho$ est la couleur (r, g, b)

Enfin on donne la sémantique des instructions :

AST	$[\![i]\!](\rho)$
`Affectation(e1,e2,a)`	$\rho[e1 \leftarrow [e2]_\rho]$. $e1$ doit être soit une variable soit un accesseur de champ
`Declaration (name,type,a)`	Cette instruction alloue `name` dans le bloc courant même s'il est déjà déclaré plus haut. `Type` doit être une expression de type valide.
`Block([i1;i2;...;ik])`	$\text{pop}([\![ik]\!](\cdots[\![i1]\!](\text{push}(\rho))))$ push et pop gèrent la pile des déclarations de l'environnement. Les variables déclarées dans le bloc sont oubliées par le pop
`IfThenElse(e,i1,i2,a)`	Si $[e]_\rho = true$, $[\![i1]\!](\rho)$ Si $[e]_\rho = false$, $[\![i2]\!](\rho)$ Non-défini sinon
`For(name,ei,et,es,body,a)`	$\rho := \rho[name \leftarrow [ei]_\rho]$ Tant que $\rho(name) <= [et]_\rho$: $\rho := [\![body]\!](\rho)[name \leftarrow [name]_\rho + [es]_\rho]$
`Foreach(name,e,body,a)`	$[e]_\rho$ doit être une liste $[v1; \cdots ; vk]$ Pour chaque vi, $\rho := [\![body]\!](\rho[name \leftrightarrow vi])$ Si name est modifié dans $[\![body]\!]$, la liste est modifiée (c'est le sens de $\leftrightarrow$).
`Draw_pixel(e,a)`	ρ. Aucun effet sur l'environnement, mais affiche le pixel $[e]_\rho$. Non-défini si $[e]_\rho$ n'est pas un pixel.
`Print(e,a)`	ρ. Aucun effet sur l'environnement, mais affiche $[e]_\rho$ au terminal.
`Nop`	ρ. Aucun effet.