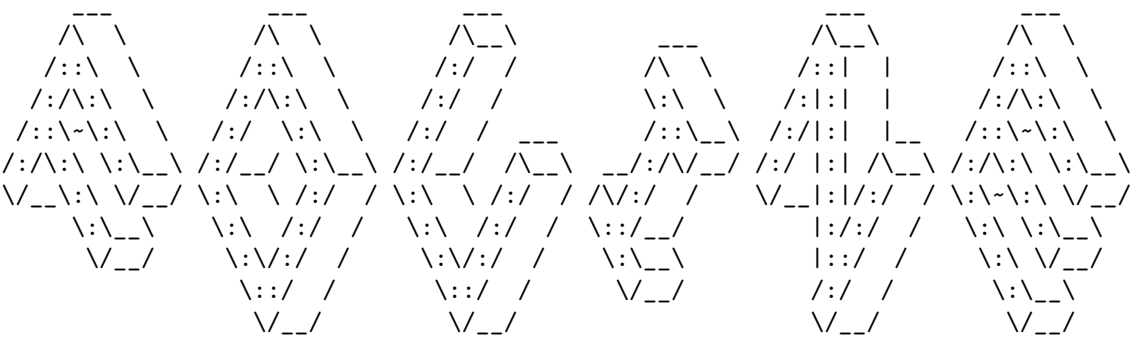

Syntaxe:

- opérateurs mathématiques simples : `+,-, /, *, =, <>, <, >, <=, >=, and, or, not`.
- Les opérateurs `+,-, /, *` sont de type `int -> int-> int`.
- `and, or, not` sont de type `bool-> bool-> bool` et `bool->bool`
- `=, <>, <, >, <=, >=` sont de type `'a->'a->int`
- structure de contrôle: `if condition then foo else bar`. `foo` et `bar` doivent avoir le même type. Les expressions du type `if cond then expr` fonctionnent également, mais `expr` doit être de type `unit`
- fonctions: `fun a-> fun b -> expr` est une fonction anonyme à deux arguments `a` et `b` évaluant l'expression `expr`
- Affectation et variables:
  - `let ident = exp in expression` est un programme affectant `expr` à l'identifiant `ident` lors de l'exécution de expression. `let ident a b c = expr in expression` est un raccourci pour l'expression `let ident = fun a-> fun b-> fun c->expr in expression`. Les expressions de la forme `let ident = expr` sont uniquement autorisées dans le toplevel.
  - `let rec ident = expr` in expression se comporte comme un `let ident = expr` à une exception près : `expr` est assigné à l'identifiant `ident` dès qu'il est vu. Cela permet de définir des fonctions récursives
- Références: les références à des valeurs de tous les types. On peut déréférencer une valeur avec `!`, en créer une avec `ref`, et changer la valeur d'une avec `:=` (type `ref 'a -> 'a -> unit'`)
- Underscore (`_`): l'underscore est implementé. Il s'agit d'un identifiant joker pouvant avoir n'importe quel valeur. Exemples:
  - `let _ = expr`

  - `let f x _ = x in f`

- Exceptions:
  - on peut renvoyer une exception avec `raise n` ou n est un entier
  - On peut récupérer des exceptions avec un bloc du type `try foo with E x -> bar`. Si `x` est une constante, `bar` est exécuté uniquement si `foo` lève une exception de numéro égale à la constante, sinon l'exception continue son chemin. Si `x` est un identifiant, `bar` est exécuté dés que `foo` lève une exception.
- array: On supporte les array d'entiers
- prInt: comme dans la spec
- ouverture de fichier: la commande `open "fichier"` ouvre le fichier `fichier`. S'il n'existe pas, ou s'il contient une erreur de parsing, le code chargé sera `()`. Attention, les chemins sont relatifs par rapport a l'endroit ou est lancé l'interpreteur, pas l'endroit ou est le fichier!
- tuples 'generalisé': on peut faire `let x, y = 1, 2`
- Types et constructeurs
  - Déclarations comme en caml avec la syntace : `type ('a, ..., 'b) nom_type = | Constr1 (of type_arguments1) .... | Constrn of (type_argumentsn)`
  - les types sont récursifs: `let 'a test = None of 'a test`
  - Les constructeurs peuvent être avec ou sans arguments.
- Pattern matching: les expressions `let 0, (), (x, _), Constr y = ....` ou `fun (x, Constr (a, b)) -> ...` sont valides
- les `;;` à la fin d'une expression sont requis
- opérateurs personnalisables. On peut redefinir un certain nombre d'opérateurs infix et préfix (@@, @, *+, |>, ...). La syntaxe est comme en caml: `let (@@) a b = ....`
- listes. On peut construire une liste vide avec `[]`, concatener des listes avec `@` et insérer un élement au début avec `::`. Elles sont compatible avec le pattern matching. Leur implémentation reposant sur les types, elles sont incompatibles avec la compilation
- Il y a plusieurs types de bases: les fonctions, les refs de quelquechose, les array d'entiers, les entiers et les booléens. `true` et `false` representent respectivement le booléen vrai et le booléen faux

Options d'interface :

L'exécutable Fouine dispose de 5 options: - debug, pour afficher le pretty print d'un fichier / commande, et d'autres informations complémentaires lorsque l'on est en mode compilateur - machine, pour passer en mode compilateur / executeur SECD - coloration, pour activer la coloration syntaxique dans les erreurs / le pretty print - inference pour activer l'inférence de types - interm pour sauvegarder le programme compilé dans un fichier - o pour enregistrer le code transformée dans un fichier annexe a destination d'être évalué par Caml. Attention cependant, les raise sont affichés comme étant Raise (E expression) ou E est une erreur non définie, mais définissable avec `exception E of int` - R, E et ER comme dans le sujet

Sans nom de fichier, fouine passera en mode repl. Sinon il exécutera le contenu du fichier selon le mode choisi (par défaut, en mode interpréteur)

Architecture:

- inference.ml contient les fonctions responsables de l'inférence de type
- inference.ml contient les fonctions responsables de l'inférence de type
- buildins.ml contient les définitions des fonctions buildins
- inference_old.ml contient les fonctions responsables de la vieille inférence de type
- transformations_ref.ml pour la transformations pour les references
- transformations_except.ml pour la transformations par les continuations
- prettyprint.ml le print d'ast fouine
- main.ml la repl et les fonctions de chargement de fichiers
- interpret.ml l'interprétation
- compilB.ml la compilation d'ast vers 'bytecode' de machine à pile SECD
- secdB.ml exécuteur de bytecode SECD
- bruijn.ml conversion en indices de De Bruijn
- dream.ml l'environnement pour la SECD et bruijn.ml
- expr.ml les types principaux de l'ast et quelques fonctions de manipulations
- env.ml, errors.ml et binop.ml sont des fichiers contenant des fonctions utilitaires
- le parser et le lexer se trouvent dans parser.mly et lexer.mll respectivement
- zinc_machine/* contient l'isa, et la machine virtuelle pour la ZINC machine (implémentée mais non fonctionnelle)

Le fichier fouine est un script bash permettant de lancer main.native avec rlwrap si cet utilitaire est ajouté

Repartition des taches:

- Pierre

  - interpréteur
  - parseur / lexer
  - inférence de types
  - main.ml (parsing des arguments, chargement de fichiers, repl...)
  - prettyprinting
  - Constructeurs & types
  - transformations des réferences et des exceptions
- Guillaume
  - transformation de l'ast vers des abstractions/indices de De Bruijn
  - compilation vers du bytecode
  - machine secd complète
  - machine zinc implémentée à partir de http://gallium.inria.fr/~xleroy/publi/ZINC.pdf, compile mais non testée pour le moment
  - script de test "testing.sh"

Implementation:

- L'interprétation se base lourdement sur les continuations: cela permet de faire aisément les exceptions, et puis au moins j'ai pu découvrir un truc
- L'inférence de type à été ajoutée pour 3 raisons principales, malgré le fait que cela ne soit pas demandé:
  - Cela permet de faire une prépass unifiée pour détecter les erreurs, commune à l'interprétation et à la compilation
  - Je n'avais jamais fait d'inférence et j'ai voulu apprendre à en faire
  - le but final est de faire du nbe, mais celui-ci à besoin de connaître le type de l'expression attendue pour fonctionner. L'inférence de type est donc une première étape vers le nbe
- La récursivité lors de la transformation par continuations se fait à l'aide des points fixes http://www.cs.cornell.edu/courses/cs3110/2013sp/supplemental/lectures/lec29-fixpoints/lec29.html
- Pour la transformation des exceptions, la variable 'globale' tr_memory contient l'état de la mémoire simulant les réfs en tout point
- Les fonctions 'buildins' (ie utilisant du code Caml, comme PrintIn par exemple) sont fonctionnelles avec l'interprétation et la compilation, et compatibles avec les transformations (même si ce point n'est pas encore clef en main et demanderait un peu de refactore). Nous ne les utilisons pas car nous ne les avons pas suffisamment testées. Elles utilisent les types BuildinClosure pour l'interpreteur et bClosure pour le compilateur
- Les opérateurs arithmétiques ne sont pas redéfinissables. Ils pourraient cependant l'être si nous utilisions les fonctions 'buildins'

Transformations et compilateur

Les transformations utilisent uniquement du code Fouine (pour gérer les environnements dans le cas de la transformation par refs, ou pour créer les points fixes pour l'autre transformations). Puisqu'elles reposent lourdement sur les types et les tuples, elles ne sont pas utilisables avec le compilateur. Nous avons en effet préféré de pas implémenter le matching dans le compilateur car la seule manière nous venant à l'esprit de manière immédiate était de passer par une fonction Caml effectuant tous le travail d'unification, ce que nous ne trouvions pas dans l'esprit de la machine à pile.

Machine à pile SECD

Environnement spécifique : module Dream

- DreamEnv est l'environnement utilisé par la SECD. Il répond à toutes les attentes définies dans l'article http://gallium.inria.fr/~xleroy/mpri/2-4/machines.pdf dont :
  - l'opération add qui incrémente d'un tous les indices des précédents éléments
  - l'opération access(n) qui accède au n-ième élément
  - la compatibilité avec les opérations de pile
- Dream est très proche et un peu moins dense et sert au renommage en indices de De Bruijn

Instruction Set

- C k, BOP op : opérations binaires
- ACCESS x n'existe plus, au profit de :
- ACC n : accède au n-ième champ de l'environnement
- CLOSURE, CLOSUREC : ne prennent en argument que code * env
- BUILTIN : implémente les fonctions builtin
- LET, ENDLET : assignation de variables dans un scope qui se termine par ENDLET, sans arguments grâce à De Bruijn
- APPLY : attrape une closure et l'applique à un argument, tous deux trouvés sur la stack
- RETURN
- PRINTIN : comme la spec
- BRANCH : choix entre deux continuations de code trouvés dans la stack
- PROG c : encapsulation de code
- REF r, BANG x : référence d'entier et de fonctions, déréférencement
- ARRAY, ARRITEM, ARRSET : gèrent les opérations sur les array
- TRYWITH, EXNCATCH : gestion des exceptions
- EXIT : arrêt de l'exécution d'un code, retour à la précédente exécution

Optimisations réalisées :

- gestion des indices de De Bruijn
- recursivité terminale

Options supplémentaires :

- compilation d'un module Fouine (plusieurs codes séparés par des ;;)
- chronomètre du temps d'exécution d'un programme (option -debug)
- implémentation de fonctions "en dur" qui réservent des identifiants clés (pas utile pour l'instant)

A venir

Implémentation des tuples et du pattern-matching. Il paraît difficile en compilation de faire du vrai matching avec des types, qui ne passe pas par des appels système dans la machine.

Machine ZINC

Trois fichiers dont une Isa détaillée dans le dossier zinc_machine. Le jeu d'instruction est celui proposé dans http://gallium.inria.fr/~xleroy/publi/ZINC.pdf. Compile mais non testée pour l'instant.

Tests:

De multiples tests sont disponibles dans le dossiers tests/ Les scripts testing.sh et testing_secd.sh sont là pour les exécuter de manière groupée. Ils prennent en argument les arguments que l'on veut faire passer à fouine. Le premier sert à tester l'interpreteur, le second la secd.

Inférence de types:

- Première version La première version de l'inférence de type est basé sur un algorithme HW lourdement modifié. Il est encore présent dans inference_old mais n'est plus compatible avec le code actuel Ci-dessous est une sorte de log de différents bugs rencontrés et des solutions utilisés

Il y a un petit hack pour matcher les constructeurs dans les expressions. On considère les constructeurs comme étant des fonctions a un argument, donc en vérité un constructeur est de la forme Call(Cosntructeur_noarg, arg) <=> Constructeur arg

Un autre hack réside dans la duplication ou nom des types quand on les récupère à partir d'un nom. Supposons qu'une fonction ref soit définie, de type 'a -> 'a ref. Avec notre système, si on récupère le type normalement par l'environnement, après l'exécution de l'expression "ref false", le type de ref devient bool -> bool ref, ce que l'on souhaite éviter: il faut donc copier le type de ref quand on l'utilise. Mais dans ce cas, que se passe-t-il quand on évalue let temp f = f 0? Le type de temp est 'a -> 'b, au lieu de (int -> 'a) -> 'b car nous avons copié le type de f avant de travailler dessus. On introduit donc un nouveau type, Arg_type, nous permettant de savoir si un type stocké vient d'un argument ou non. Un argument ne peut qu'être spécialisé, nous ne devons pas le copier, alors qu'un type défini par un let normal ne peut pas être plus spécialisé. Mais un autre problème se lève avec une expression de la forme: let f e = let (x, y) = e in x (qui a alors le type 'a * 'b -> 'c au lieu de 'a * 'b -> 'a). En effet, en récupérant x, on copie son type. Mais x a un type 'a! Or en caml aucun identifiant valable ne peut avoir de type 'a: on ne doit donc pas copier son type.

Autre bug (cité dans un commit): Pour le typage de fibo (let fibo n = let rec aux a b i = if i = n then a else aux b (a+b) (i+1) in aux 0 1 0;;), le typage est mal fait et était 'a -> int (avant le fix avec leshasmaps). En effet, lors de l'unification, si on unifie un 'a avec un autre type, on unifie ce 'a en particulier (il n'y a pas vraiment de pointeurs en caml, et même en C++ la tache serait non triviale: on veut, étant donné a1,...,an pointant vers le même objet, et b1, ..., bm pointant vers un même autre objet, faire pointer les ai et bi vers le même objet et ainsi de suite). Pour contrer ce problème, on ajoute dans une hashmaps ces affectations (du type 'a := int), puis dans un postprocess, on résout les 'a non affecté à l'aide de ce hashmaps (on parcours le type, si on voit un 'a on regarde s'il a été affecté), et on itère cette procédure tant que quelque chose bouge (car on pourrait introduire d'autres 'b non correctement unifiés).

- Seconde version: L'implémentation de la première version de l'inférence devenant peu lisible au fil des bugfixs, et étant encore extrêmement buggée (et indebuggable), il a été décidé de la réécrire en suivant le lien suivant: http://okmij.org/ftp/ML/generalization.html Ce site propose un algorithme évitant plusieurs problèmes rencontrés précédemment manifestement plus lisible.

Types:

Pour implémenter proprement les types, le pattern matching et les points fixes, il a été décidé d'implémenter un systême de constructeurs. Pour déclarer les types la syntaxe est identique au caml: `type ('a, ..., 'b) nom_type = | Constr1 (of type_arguments1) .... | Constrn of (type_argumentsn)` Les types peuvent être récursifs.

Les Constructeurs en eux mêmes sont délicats à parser. En effet, une expression comme Constr a b pourrait être potentiellement comprise lors du parsing comme

(Constr) a b ou (Constr a) b. Pour résoudre ce parsing, on dispose de trois résultats possibles après le parsing: - Constructeur_noarg(nom_constructeur, _) -> constructeurs sans argument - Constructeur(nom_constructeur, arguments, _) -> constructeurs avec argument dans une zone d'affectation (pour les expressions comme `let Constr x =...` ou `fun Constr x -> ....`) - Call(Constructeur_noarg(nom_constructeur, _), arguments, _) qui est équivalent à Constructeur(nom_constructeur, arguments, _) -> le reste

Sans inférence de type, on ne vérifie même pas si un constructeur est bien défini.

A cela s'ajoute également du pattern matching

Issues:

- Si les transformations sur les exceptions sont activés, certains letrecs ne sont pas bien inférés avec notre inférence comme avec l'inférence caml:
  - `let rec fact n = if n = 0 then 1 else n * fact (n-1);;`
  - `let rec fact n = if n = 0 then 1 else n * fact (n-1) in fact;;` C'est étrange car `let rec fact n = if n = 0 then 1 else n * fact (n-1) in fact 8;;` est correctement typé. Nous ne savons pas du tout d'ou vient ce bug
- La maniére dont nous gérons les LetRecs présent dans le scope global avec la transformation par continuation n'est pas optimale. Ainsi, des expressions de la forme `let rec test x = test x + 1 ;;` (typage cyclique) sont mal typés alors que `let rec test x = test x + 1 in test 3;;` l'est bien Ce bug empêche par exemple la définition de `@` dés que la transformation -E est activée