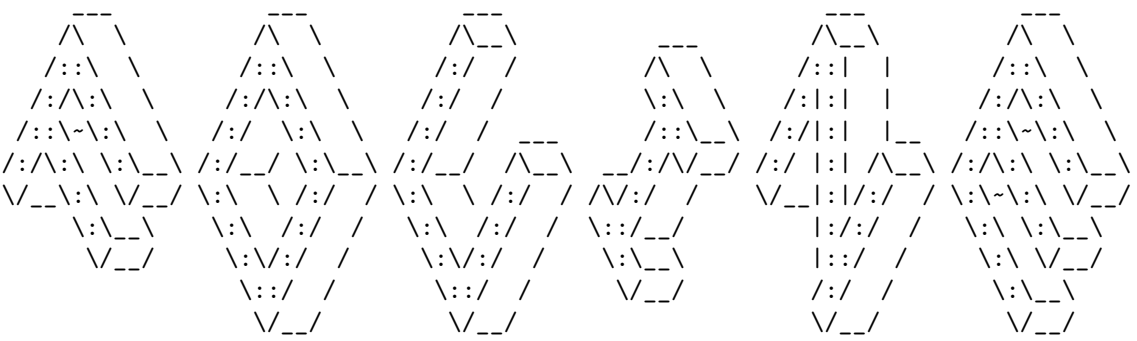

## Syntaxe:

- 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 `()`
- les `;;` à la fin d'une expression sont requis

## 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

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

repl compile -> pas de sauvegarde d'environnement car n'a pas vraiment de sens

## Architecture:

- inference.ml contient les fonctions responsables de l'inférence de type
- prettyprint.ml le print d'ast fouine
- main.ml la repl et les fonctions de chargement de fichiers
- interpret.ml l'interprétation
- compil.ml la compilation d'ast vers 'bytecode' de machine à pile SECD
- secd.ml exécuteur de bytecode SECD
- 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

Le fichier fouine est un script bash permettant de lancer main.native avec rlwrap si cet utilitaire est ajouté ### Expérimental - zinc.ml contient un début de compilateur vers la machine ZINC (manque de temps pour finir l'implémentation)

## Repartition des taches:

- Pierre
  - interpréteur
  - parseur / lexer
  - inférence de types
  - main.ml (parsing des arguments, chargement de fichiers, repl...)
  - prettyprinting
- Guillaume
  - compilation de l'ast vers du 'bytecode'
  - machine secd complète (toutes extensions sauf ref de fonctions)
  - ebauche de machine zinc

## Implementation(Pierre):

- 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 ajouté pour 3 raisons principales, malgré le fait que cela ne soit pas demandé:
  - Cela permet de faire une prépass unifié 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

## Machine à pile SECD

- C k, BOP op : opérations binaires
- ACCESS(x) (pas encore ACCESS(n))
- UNITCLOSURE, CLOSURE, CLOSUREC : permettent de gérer respectivement les fonctions à argument unit/underscore, les fonctions courantes et les fonctions récursives. CLOSUREC a pour particularité d'encapsuler un environnement qui contient une CLOSURE identique à elle-même
- LET x, ENDLET : assignation de variables dans un scope qui se termine par ENDLET
- 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, déréférencement
- ARRAY, ARRITEM, ARRSET : gèrent les opérations sur les array
- TRYWITH : gestion des exceptions
- EXIT : arrêt de l'exécution d'un code, retour à la précédente exécution

Ce que j'aimerais faire :

- passer aux indices de De Bruijn
- finir l'implémentation de la ZINC machine