TELECOM SudParis

Projet Informatique 1ère Année
CSC 3502

CAMELRAM BÊTA

ALUSH YASEMINE
AMSALLEM CLÉMENT
LAGARRIGUE LÉO
PETIT CHABANE
WINNINGER THOMAS

Enseignant responsable : Pascal HENNEQUIN

30 Mai

Table des matières

Chapitre 1

Introduction

Mais où sont donc passé les mathématiques à TSP ? Étant en manque d'algèbre et d'informatique théorique nous avons décidé de reprendre nos cours de prépa et d'y ajouter nos nouvelles connaissances pour créer un outil mêlant mathématique et informatique. C'est donc pour cela que nous nous sommes dirigés vers l'implémentation d'une calculatrice formelle en OCaml. C'est l'occasion pour nous d'implémenter un interpréteur, ainsi que des structures de données intéressantes comme des arbres.

En plus de l'aspect interprétation, une interface graphique sera implémentée, permettant à l'utilisateur de rentrer des formules et de recevoir un résultat sous forme LaTeX. Les algorithmes utilisés seront aussi étudiés pour être rapides et efficaces. Il s'agit de trouver des méthodes de calcul optimisées pour réduire le temps d'exécution et l'espace mémoire utilisé.

Chapitre 2

Cahier des charges

La calculatrice devra être simple et agréable à utiliser. Pour cela une interface graphique a été choisie. Elle devra être interactive et comporter :

— Un espace pour entrer les commandes et les formules
— Une interface de saisie pour ajouter des caractères spéciaux
— Un espace pour visualiser le résultat sous la forme LaTeX

Les échanges entre les différents modules seront standardisés pour pouvoir rajouter des fonctions à la calculatrice à tout moment. Il faut :

— Une norme pour échanger les données sous format de chaîne de caractères : Cette norme a été choisie, ce sera une variation du langage LaTeX

La calculatrice devra pouvoir effectuer des intégrations (calculs approchés), dérivations, transformées de Fourier et des opérations sur les polynômes. Les opérations devront être optimisées en temps et en espace. Plus précisément les attentes sont :

— Opérations sur les polynômes (multiplication, Bézout)
— Dérivation de fonctions
— Intégration en utilisant des approximations
— Transformée de Fourier

Le langage choisi pour les algorithmes de calcul est le OCaml 5.0

Chapitre 3

Développement

3.1 Conception préliminaire

3.1.1 Interface graphique

Pour ce qui est de l'interface, on aura à priori :
— GTK pour créer une UI sur un framework simple et léger, écrit en C
— GTKMathView pour l'intégration des formules mathématiques dans l'UI → test d'intégration à prévoir
— Un programme permettant de procéder à la traduction de l'input utilisateur en MathML (le langage utilisé étant encore à définir, ce serait de préférence du bash ou du → idem
— Un autre programme se chargeant de retraduire la formule MathML en formule quelque chose d'utilisable par l'interpréteur → idem

3.1.2 Mathématiques

Pour ce qui est de l'aspect mathématique, les algorithmes choisis sont ceux du CIRM [1] :
— Multiplication : multiplication rapide avec interpolation et algorithme de Horner
— Division euclidienne : pseudo-division pour les polynômes dans un anneau particulier
— PGCD
— Factorisation : algorithme de Gauss et de Berlekamp, tests de primalité
— Intégration numérique : méthode de Simpson avec interpolation et méthode de Monte-Carlo
— Dérivation : algorithme de Richardson
— Simplification d'expression : algorithme de Gröbner

3.1.3 Interpréteur

Pour l'aspect compilation, mise en place d'un algorithme en OCaml qui va interpréter (analyse descendante) l'entrée utilisateur pour transformer l'expression en un arbre. Arbre qui sera ensuite analysé pour appeler les différents programmes concernés.

3.2 Conception détaillée

3.2.1 Interface graphique

Au final, le choix de GTK n'ayant pas été retenu du fait de l'âge des librairies disponibles qui rendait leur documentation inaccessible ou incorrecte, on aura fait le choix de Qt pour la construction de

l'interface. Elle se compose de deux fenêtres, de classe respectives MainWindow et VarEditWindow (héritées respectivement de QMainWindow et QWidget). La première correspond à la fenêtre principale, la deuxième à une fenêtre qui permet de visualiser les variables actuellement enregistrées.

MainWindow

Cette classe est composée de deux fonctions publiques :

— Son constructeur, qui connecte une partie des boutons de la barre de menu avec 3 fonctions privées qui sont :
  — int clearDisplay() -> appelle displayLatex("") et vide le champ d'entrée utilisateur
  — int varReset() -> vide toutes les variables
  — varManage() -> ouvre une nouvelle fenêtre qui permet d'observer les variables rentrées jusque là après avoir mis à jour la liste des variables dans ladite fenêtre.

  Il fait également afficher à mathjaxView (le widget webkit) un fichier html de bienvenue et permet de prendre la forme de l'IU à partir du fichier mainwindow.ui, qui contient aussi des informations sur la manière dont communiquent certains items du menu.
— Son destructeur, qui ne fait que supprimer l'ui.

Ainsi que de six autres fonctions privées :

— int displayText(const QString &inputText) -> incorpore à un fichier html où MathJax est chargé du texte à afficher
— int displayLatex(const QString &inputLatex) -> appelle displayText("
("+inputLatex"
)")
— int on_displayExpr_clicked() -> met l'entrée utilisateur dans la variable latek et appelle displayLatex(latek) puis remet le focus sur le champ d'entrée utilisateur.
— int on_disCompExpr_clicked() ->
  — met l'entrée utilisateur dans la variable latek
  — isole les variables pour les rajouter dans le tableau variables
  — vérifie qu'aucune variable ne prend la même valeur deux fois.
  — appelle une instance de l'interpréteur en passant toutes les déclarations de variable contenues dans variables
  — si l'instance de camelrambeta affiche quelque chose dans le stderr, l'information est relayée avec displayText
  — sinon le contenu du stdout est affiché avec displayLatex()
  — remet le focus sur le champ d'entrée utilisateur
— QStringList splitEqu(const QString &input) -> sépare les déclarations de variables entre elles et aussi du calcul principal
— int on_clearExpr_clicked() -> vide le champ d'entrée utilisateur

VarEditWindow

Cette classe est composée elle aussi de deux fonctions publiques qui sont

— Son constructeur, qui ne fait que prendre le modèle d'UI à partir de vareditwindow.ui.
— Et son destructeur, qui lui aussi ne fait que supprimer l'ui.

Ainsi que d'une fonction privée "int updateList(const QStringList &vars)", qui permet de mettre à jour la liste des variables.

3.2.2 Mathématiques

- Les algorithmes optimisés seront utilisés pour les opérations sur les polynômes. Le reste des opérations sera fait avec des algorithmes récursifs en essayant de réduire le plus possible leur complexité.
- Pour la multiplication rapide de polynômes, nous avons choisi l'algorithme de Karatsuba.
- Pour l'approximation d'intégrales, nous avons choisi de traiter à part le cas des bornes infinies. En effet, on effectuera d'abord un changement de variables $x \mapsto \frac{1}{x}$ afin de ramener l'intervalle $[1, +\infty]$ sur $[0, 1]$, de même pour les négatifs.

3.2.3 Interpréteur

Pour la partie interprétation, nous avons choisi d'utiliser le module Menhir de OCaml, qui est un descendant de ocamlyacc. Ce module facilite la création du lexeur, du parseur et de la création de l'arbre syntaxique. Pour cela, on utilise les fichiers `lexer.mly`, `parser.mly` et `ast.ml` afin d'y mettre les règles syntaxiques. Il suffit ensuite de faire appel au module qui s'occupe de créer les fichiers `.ml` correspondants.

AST

Voici la structure de notre AST :

```
ast.ml
(**
    TYPE expr
    Type of the Abstract Syntax Tree
*)
type expr =
    | EVar              of string
    | EInt              of int
    | EFloat            of float
    | EBop              of bop * expr * expr
    | EUop              of uop * expr
    | EIntegral         of expr * expr * expr * expr
    | EIntegralImplicit of expr * expr * expr
    | EDifferentiate    of expr * expr
    | EPol              of string * expr
    | EPolImplicit      of expr
    | ELet              of string * expr * expr
    | EEval             of expr
    | ESimplify         of expr
```

`EBop` et `EUop` désignent les opérateurs binaires et unaires qui sont données à part, car ils seront traités de la même façon :

```
ast.ml
(**
    TYPE bop
    Binary Operator, used to simplify pattern matching
*)
type bop = BAdd | BSub | BMul | BDiv | BPow

(**
    TYPE uop
    Unary Operator, used to simplify pattern matching
*)
type uop =  UMinus | UExp | ULog | UCos | USin | UTan
    | UAcos | UAsin | UAtan | UCosh | USinh | UTanh
    | UCeil | UFloor | URound
```

Ainsi, une expression comme

$$3 * X + e^{\frac{X}{\sin(X)}}$$

s'écrit :

```
EBop(BAdd, EBop(BMul, EInt 3, EVar "X"), EUop(UExp, EBop(BDiv, EVar "x",
EUop(USin, EVar "x"))))
```

Ou encore, une expression avec déclaration de variable

$$\text{let} \quad X = \int_{-\infty}^{0} Y^2 \quad \text{in} \quad X + 2$$

s'écrit :

```
ELet(EVar "X", EIntegralImplicit(EUop(UMinus, EVar "infty"), EInt 0,
EBop(BPow, EVar "Y", EInt 2)), EBop(BAdd, EVar "X", EInt 2))
```

Les différences entre `EIntegralImplicit, EPolImplicit` et `EIntegral, EPol` résident dans le choix de l'utilisateur de préciser ou non les variables muettes.
Cet arbre sera ensuite parcouru grâce à des fonctions récursives afin d'effectuer les opérations demandées par l'utilisateur. Ces opérations (à part celles d'affichage) se trouvent dans le fichier `calc.ml`

Lexeur

Grâce au module menhir, nous pouvons construire le lexeur simplement en listant les règles à appliquer. En voici un exemple :

lexer.mll

```
let letter = ['A'-'Z'] | ['a'-'z']
let digit = ['0'-'9']
let non_digit = ['_' '\\'] | letter
let ident = non_digit+

let line_comment = "//" [^ '\n']*

rule token = parse
    |  [' ' '\t'] | line_comment
        { token lexbuf }
    | ['\n']
        { Lexing.new_line lexbuf; token lexbuf }
    | ident as str
        {
            match str with
                | "\int_" -> INTEGRAL
                | "let" -> LET
                | "in" -> IN
                | "\exp" -> EXP
                | "\round" -> ROUND
                | "\infty" -> INFTY
                | "\pi" -> PI
                | "\frac" -> FRAC
                | s -> IDENT(s)
        }
    | ['=']
        { EQUAL }
    | ['+']
        { ADD }
    | ['*']
        { MUL }
```

```
    | ['^']
        { POW }
    | digit+ as lxm
        { INT(int_of_string lxm) }
    | ['{']
        { LCBRA }
    | ['}']
        { RCBRA }
    | eof
        { EOF }
    | _ as c
        { raise (Error c) }
```

Le lexeur va donc créer une liste de tokens en fonction de ce qu'il reçoit en entrée. Ces tokens vont ensuite être analysés par le parseur qui va créer l'AST.

Parseur

De nouveau grâce au module menhir, il suffit de lister les règles syntaxiques voulues, on cherche d'abord à définir les règles de prédominance des opérateurs :

```
parser.mly
%left "+" "-"
%left "*" "/"
%left "^"
%left "."
%nonassoc UMINUS, EXP, ROUND
```

Il faut ensuite donner les règles à appliquer en cascade :

```
parser.mly
main:
    | e = let_expr EOF
        { e }

let_expr:
    | LET id = IDENT "=" e1 = let_expr IN e2 = let_expr
        { ELet(id, e1, e2) }
    | e = expr
        { e }

expr:
    | i1 = INT "." i2 = INT
        { EFloat(float_of_string ((string_of_int i1) ^ "." ^
            (string_of_int i2))) }
    | i = INT
        { EInt(i) }
    | INTEGRAL e1 = expr "^" e2 = expr e3 = expr
        { EIntegralImplicit(e1, e2, e3) }
    | e1 = expr "+" e2 = expr
        { EBop(BAdd, e1, e2) }
    | e1 = expr "*" e2 = expr
        { EBop(BMul, e1, e2) }
    | e1 = expr "^" e2 = expr
        { EBop(BPow, e1, e2) }
```

```
    | "-" e = expr %prec UMINUS
        { EUop(UMinus, e) }
    | EXP e = expr %prec EXP
        { EUop(UExp, e) }
    | ROUND e = expr %prec ROUND
        { EUop(URound, e) }
    | x = IDENT
        { EVar(x) }
    | x = INFTY
        { EVar("infty") }
    | "{" e = let_expr "}"
        { e }
```

Analyse syntaxique et calculs

Une fois l'arbre syntaxique obtenu, les calculs sont répartis dans plusieurs fichiers. Le plus important est `calc.ml`, c'est celui qui concentre les opérations principales sur l'arbre. Les fonctions sont toutes du même type : des fonctions récursives qui vont filtrer les noeuds de l'arbres. En voici un exemple avec la fonction qui s'occupe de calculer la dérivée d'une expression :

calc.ml

```
(**
    FUNCTION differentiate
    @type val differentiate : expr -> string -> expr = <fun>
    @returns The derivative of expr with respect to the variable given as
        argument
*)
let rec differentiate expr x=
    match expr with
        | EDifferentiate(e1, y) -> EDifferentiate((differentiate e1 x), y)
        | EBop(op, e1, e2) ->
            begin
                match op with
                    | BAdd -> EBop(BAdd, (differentiate e1 x),
                        (differentiate e2 x))
                    | BSub -> EBop(BSub, (differentiate e1 x),
                        (differentiate e2 x))
                    | BMul ->
                        EBop(BAdd, EBop(BMul, (differentiate e1 x), e2),
                        EBop(BMul, e1, (differentiate e2 x)))
                    | BPow ->
                        EBop(BAdd, EBop(BMul, EBop(BMul, EUop(ULog, e1),
                        EBop(BPow, e1, e2)), (differentiate e2 x)),
                        EBop(BMul, EBop(BMul, e2, EBop(BPow, e1,
                        EBop(BSub, e2, EInt(1)))), (differentiate e1 x)))
            end
        | EUop(op, e1) ->
            begin
                match op with
                    | UMinus -> EUop(UMinus, (differentiate e1 x))
                    | UExp ->
                        EBop(BMul, (differentiate e1 x), EUop(UExp, e1))
                    | URound ->
                        failwith "ERREUR : Derivation de fonction non
                        derivable"
            end
```

```
        | EFloat(_) -> EInt(0)
        | EInt(_) -> EInt(0)
        | EVar(y) when y = x -> EInt(1)
        | EVar(y) -> EInt(0)
```

Les fonctions atomiques sont regroupées dans le fichier `general.ml`. De base, la conception devait passer par des functors. En OCaml, les functors (foncteurs) sont des constructions qui permettent de créer des modules génériques et réutilisables. Les functors permettent de paramétrer un module par un autre module, en fournissant une interface et une implémentation spécifiques. Malheureusement, nous n'avons pas eu le temps de nous plonger dans l'utilisation de ces objets. Nous sommes donc restés avec des types simples, l'implémentation des fonctions de base nécessite donc de filtrer sur toutes les possibilités, ce qui n'est pas optimal, mais qui est du moins fonctionnel. En voici un exmple avec la fonction addition :

```
general.ml
(**
    FUNCTION add
    @type val add : value -> value -> value = <fun>
*)
let add a b =
    match a, b with
        | VInt(x), VFloat(y) -> VFloat(float_of_int x +. y)
        | VInt(x), VInt(y) -> VInt(x + y)
        | VFloat(x), VFloat(y) -> VFloat(x +. y)
        | VFloat(x), VInt(y) -> VFloat(float_of_int y +. x)
```

Enfin, il y a trois modules pour les trois objets spéciques : les fonctions, les variables et les polynômes qui sont traités à part. Pour les variables, le module est classique, il permet une simple encapsulation.

```
variable.ml
module Variable = struct
    open Map
    open Ast

    module Dict = Map.Make(String)

    let variables = ref Dict.empty;;
    variables := Dict.add "infty" (VFloat (Float.infinity)) !variables

    (**
        FUNCTION Variable.to_string
        @type val to_string : value -> string option = <fun>
        @returns Some x if the variable exists, None if it doesn't
    *)
    let to_string = function
        | None -> "La variable n'existe pas."
        | Some(s) -> match s with
            | VInt(x) -> string_of_int x
            | VFloat(x) -> string_of_float x

    (**
        FUNCTION Variable.get
        @type val get : string -> value option = <fun>
        @returns Some x if the variable exists, None if it doesn't
```

```
    *)
    let get key =
        try Some(Dict.find key !variables) with
            Not_found -> None

    (**
        FUNCTION Variable.add
        @type val add : string -> value -> unit = <fun>
    *)
    let add name value =
        variables := Dict.add name value !variables
```

En travaillant sur le projet, nous avons eu l'occasion de nous familiariser avec les modules, et de comprendre comment ils s'utilisent en OCaml. Comme le Java, le OCaml possède une notion d'interface et de classes abstraites. Pour gérer l'encapsulation, on peut définir une interface dans le même fichier que le module. Cela permet de déterminer qu'est ce qui est 'public' et qu'est ce qui reste 'privé'. En voici un exemple d'implémentation avec le module `polynomial.ml` :

```
polynomial.ml
module type Polynomial = sig
    exception BadType
    val add : Ast.expr -> Ast.expr -> Ast.expr
    val simplify_frac : Ast.expr -> Ast.expr -> Ast.expr
    val mult : Ast.expr -> Ast.expr -> Ast.expr
end

module Polynomial : Polynomial = struct
    open Ast
    open General
    exception BadType

    type polynomial = (value * int) list

    (**
        FUNCTION pol_of_expr
        @type val pol_of_expr : string -> expr -> polynomial = <fun>
        @returns The polynomial created from the expr, with the dummy
            variable given as argument
    *)
    let pol_of_expr (variable : string) (expression : expr) : polynomial =
        let rec aux var acc = function
            | EBop(BAdd, EInt a, b) -> aux var ((VInt a, 0) :: acc) b
            | EBop(BAdd, EFloat a, b) -> aux var ((VFloat a, 0) :: acc) b
            | EBop(BMul, a, EVar var) -> (val_of_expr a, 1) :: acc
            | EBop(BMul, a, EBop(BPow, EVar var, EInt b)) ->
                (val_of_expr a, b) :: acc
            | EBop(BAdd, a, b) -> (aux var acc a) @ (aux var acc b)
            | _ -> raise BadType
        in aux variable [] expression

    (**
        FUNCTION expr_of_pol
        @type val expr_of_pol : string -> polynomial -> expr = <fun>
    *)
    let rec expr_of_pol var = function
```

```
        | [] -> EInt 0
        | [(a, b)] -> EBop(BMul, expr_of_val a, EBop(BPow, EVar var,
            EInt b))
        | x :: q -> EBop(BAdd, expr_of_pol var [x], expr_of_pol var q)

    (**
        FUNCTION add_poly
        @type val add_poly : polynomial -> polynomial -> polynomial
            = <fun>
    *)
    let rec add_poly poly1 poly2 =
        match (poly1, poly2) with
        | [], _ -> poly2
        | _, [] -> poly1
        | (coeff1, exp1) :: rest1, (coeff2, exp2) :: rest2 ->
            if exp1 = exp2 then
                let sum = General.add coeff1 coeff2
            in
                if sum = VInt 0 || sum = VFloat 0.0 then
                    add_poly rest1 rest2
                else
                    (sum, exp1) :: add_poly rest1 rest2
                else if exp1 > exp2 then
                    (coeff1, exp1) :: add_poly rest1 poly2
                else
                    (coeff2, exp2) :: add_poly poly1 rest2

    (**
        FUNCTION mult_scal_poly
        @type val mult_scal_poly : polynomial -> polynom=mial ->
            polynomial = <fun>
    *)
    let mult_scal_poly scalar poly =
        List.map (fun (coeff, exp) ->
                (General.mult coeff scalar, exp)
            ) poly
end
```

3.2.4 Fonctionnement général de l'interface graphique

Structure du projet

```
assets
camelram-beta-gui.pro
main.cpp
mainwindow.cpp
mainwindow.h
mainwindow.ui
vareditwindow.cpp
vareditwindow.h
vareditwindow.ui
```

assets

Ici, le dossier assets contient la librairie MathJax, utilisée pour interpréter le LaTeX, ainsi que les templates html utilisées et le logo du projet.

mainwindow.*

Les fichiers liés à la création de la clase MainWindow ainsi qu'a la mise en place de ses méthodes et la mise en forme de son interface.

vareditwindow.*

Les fichiers liés à la création de la clase VarEditWindow ainsi qu'a la mise en place de ses méthodes et la mise en forme de son interface.

Fonctionnement général

Ce programme vise à passer le LaTeX fourni par l'utilisateur à l'interpréteur, en extraire les déclarations de variables pour pouvoir afficher l'expression seule et en dessous l'expression retournée par l'interpréteur.

L'instance w de MainWindow se charge de l'affichage de la fenêtre principale avec laquelle l'utilisateur va intéragir en priorité. Elle pourra aussi faire apparaître l'instance d de VarEditWindow lorsque l'utilisateur voudra consulter les variables qu'il a rentrées jusque là.

3.2.5 Fonctionnement général de l'interpréteur

Structure du projet

```
ast.ml
_build
calc.ml
exec_camelrambeta
function.ml
general.ml
lexer.mll
LICENSE
main.ml
main.native
Makefile
parser.mly
polynomial.ml
README.md
tests.ml
test.txt
variable.ml
```

Le projet ne comporte pas de dossiers car la compilation des fichiers est faite à la main avec `ocamlc` et `ocamlbuild` et ne permet pas de créer d'arbre de fichers trop complexes. L'objectif était de passer sur une structure de compilation plus performante : `dune`, qui est le successeur de `ocamlbuild`. Cependant, nous n'avons pas eu le temps de passer à ce nouveau système pour plusieurs raisons. Le fonctionnement de `dune` demande de bien connaître le système de modules et d'interfaces en OCaml, ce qui n'était pas le cas au lancement du projet. La deuxième raison est plus technique, n'ayant pas d'IDE correct pour programmer en OCaml (le plugin OCaml sur Intellij n'ai pas mis à jour et celui sur VS-code ne fonctionne pas correctement avec `menhir`), nous avions été obligé de faire un `Makefile` à la main, pour pouvoir faire tous nos tests et notre compilation. Pour cela, l'ancien système d'`ocamlbuild`

était bien plus facile à utiliser.
Le projet contient une `LICENSE` et un `README.md`, mais aussi un dossier `_build` où sont stockés tous les fichiers nécessaires à la compilation de `main.ml`. Ce fichier `main.ml` est le début et la fin de l'executable `camelrambeta` crée. C'est lui qui s'occupe de récupérer la chaîne de caractère LaTeX et de renvoyer la chaîne LaTeX voulue. Le voici :

main.ml
```
(**
    FUNCTION main
    @requires File in stdin
    @type val main : unit -> unit = <fun>
    @returns Evaluation of the LaTeX string in stdin
*)
let main =
    let lexbuf = Lexing.from_channel stdin in
    let res =
        try Parser.main Lexer.token lexbuf
        with
        | Lexer.Error c ->
            fprintf stderr "ERREUR : Lexeur, ligne %d: caractere '%c'\n"
                lexbuf.lex_curr_p.pos_lnum c;
            exit 1
        | Parser.Error ->
            fprintf stderr "ERREUR : Parseur, ligne %d:\n"
                lexbuf.lex_curr_p.pos_lnum;
            exit 1
    in
        Printf.printf "%s\n" (pprint_value (eval (split_var res)))
```

Il essaye aussi de récupérer deux erreurs liées au lexing et au parsing.

Fonctionnement général

Ce programme vise à compiler des parties du LaTeX . Il y a :
— Les opérations basiques sur les entiers et flottants : `+ - * / ^`
— Les opérations basiques sur les flottants :
`\exp \log \cos \sin \tan \acos \asin \atan \cosh \sinh \tanh \ceil \floor \round`
— L'approximation d'intégrales : `\int_0^1(x)`
— La possibilité d'utiliser des variables locales : `let x = 2^64 in`
— La dérivation d'expression
— La simplification d'expression -> Ne marche pas
— Des opérations optimisés sur les polynômes -> N'est pas encore intégré au projet

Exemples fonctionnels :

$$\log(e^2)^3 - 2^\pi$$

```
camelrambeta "\log(\exp(2))^3-2^(\pi)"
> -0.824978
```

$$\int_0^{\frac{1}{\pi}} \int_{-\infty}^{+\infty} \frac{sin(x)}{x} dx \int_{y+1}^{0} \frac{1-e^{zy}}{z} dz dy$$

```
camelrambeta "let a = \int_{-\infty}^{\infty}{\sin(x)/x}/\pi in \int_0^a
{\int_{y+1}^0{(1-\exp(z*y))/z}d(z)}d(y)"
> 1.228285
```

Makefile

Une partie non-négligeable du projet a été les tests. Nous avons été confronté à un gros problème, pour pouvoir tester notre code correctement, il faut passer par le `top-level` de OCaml qui est accessible via `utop`. Cependant, utop est fait pour tester des petits scripts et non des projets en entier. Pour cela il faut d'abord complier les librairies que l'on veut utiliser correctement, compiler le code à tester à part, et donner le tout avec un fichier d'initialisation à `utop` lors de son lancement. N'ayant pas d'outil (plugin dans un IDE, logiciel...) nous sommes passés par un `Makefile` qui fait tout à la main. Ayant trouvé très peu de documentation sur utop, c'est un peu brouillon, mais c'est fonctionnel, cela permet de tester correctement et rapidement lors de la production.

```
LIB = ast
TEST = calc

all:
	ocamlbuild -use-menhir main.native
	cat test.txt | ./main.native

path:
	ocamlbuild -use-menhir main.native
	sudo cp main.native /usr/local/bin/camelrambeta_pipe
	sudo cp exec_camelrambeta /usr/local/bin/camelrambeta

.SILENT:
clear:
	ocamlbuild -clean

.SILENT:
test: clear all
	echo "#directory \"_build\";;" > init_utop.ml
	for l in $(LIB) ; do \
		ocamlc -c $$l.ml -o _build/$$l.cmo -I _build/ ; \
		echo "#load \"$$l.cmo\";;" >> init_utop.ml ; \
	done
	for t in $(TEST) ; do \
		ocamlc -c $$t.ml -o _build/$$t.cmo -I _build/ ; \
		echo "#use \"$$t.ml\";;" >> init_utop.ml ; \
	done
	utop -init init_utop.ml
	rm -f init_utop.ml

unit:
	ocamlbuild -use-menhir main.native > /dev/null
	ocamlbuild -clean > /dev/null
	ocamlbuild tests.native > /dev/null
ifneq ("$(wildcard $(tests.native))", "")
	@echo "ERREUR : ocamlbuild n'a pas reussi a compiler tests, essayez
        'make test LIB=\"ast variable function general calc\" TEST=tests'"
else
	./tests.native
	rm tests.native
endif
```

Les commandes sont :

- `make` : Pour compiler le tout et tester le programme avec la string LaTeX contenue dans `test.txt`
- `make path` : Pour compiler le tout et mettre un executable dans `/usr/local/bin/camelrambeta`, executable qui permettra de tester les exemples comme montré plus haut. (Cette commande nécessite les droits d'écrire dans `/usr/local/bin`, il faut aussi que ce chemin soit dans le `PATH` pour pouvoir utiliser la commande `camelrambeta` partout)
- `make test` : Pour faire des tests sur le code qu'on est en train de développer, par exemple, si on travaille sur `playground.ml` qui nécessite `ast.ml` et `calc.ml`, on peut faire : `make test LIB="ast␣calc"TEST="playground"`
- `make unit` : Pour effectuer tous les tests unitaires présents dans `tests.ml`

3.3 Codage

Tout le code est présent sur GitHub : github.com/Camelram-Beta/compiler et github.com/Camelram-Beta/frontend-app, le dépot est privé, il faut en demander l'accès à `thomas.winninger@telecom-sudparis.eu` par exemple.
Pour la première version du livrable 3, les polynômes sont mis à part et ne marchent pas dans le projet. La dérivation et la simplification sont mis à part. Ce qui reste est fonctionnel, il sert à évaluer le LaTeX en entrée. Les parties vont être mises ensembles au plus vite.

3.4 Tests unitaires

3.4.1 Interface graphique

Du fait de la nature du programme, un script similaire à celui implémenté pour l'interpréteur est pratiquement impossible ou du moins extrêmement peu pratique. De ce fait, aucun test unaire n'a été réalisé régulièrement, si ce n'est par l'utilisation du stdout via le flux qDebug() pour vérifier le comportement d'une fonction pour une entrée en cas limite donnée par le biais du champ de saisie utilisateur.

3.4.2 Interpréteur

Un script a été implémenté pour automatiser des tests. Nous n'avons malheureusement pas pu l'utiliser durant le projet par contrainte de temps, mais il est fonctionnel et peut être utilisé pour conclure le projet et voir ce qui à vraiment fonctionné. Le voici :

```
let test_message x y m i =
    if x = y then
        (Printf.sprintf "TEST␣%i␣:␣Pass" i, true)
    else
        (Printf.sprintf "TEST␣%i␣:␣Fail␣%s" i m, false);;

let test x y i = test_message x y "␣" i;;

let tests = [
    test (vars (EBop(BAdd, EVar "x", EInt 3))) ["x"];
    test (floor (VFloat 3.14)) (VFloat 3.);
]

let () =
    let rec aux b i = function
        [] ->
            if b then
                print_endline "***␣FINISHED␣:␣OK␣***"
```

```
            else
                print_endline "*** FINISHED : FAIL ***"
        | x :: q ->
            begin
                match x i with
                    y, z -> print_endline y; aux (z && b) (i + 1) q
            end
    in
        print_endline "*** START ***";
        aux true 0 tests;;
```

La syntaxe est assez simple, il suffit de rentrer les tests sous forme d'une liste dans `tests`, la syntaxe est `test x y`, pour vérifier si `x` est bien égal à `y`. Nous n'avons pas non plus eu le temps de faire des tests d'intégration.

Chapitre 4

Manuel utilisateur

4.1 Compiler les sources soit même

-> jsp quoi dire, c'est pas ouf ça -> compiler les sources de l'interpréteur(optionnel) (par pitié fais des sections réservées à l'interpréteur)

4.1.1 Installer Opam

Ubuntu

```
add-apt-repository ppa:avsm/ppa
apt update
apt install opam

eval $(opam env)

opam install utop ocamlbuild
```

Archlinux

```
pacman -S opam

eval $(opam env)

opam install utop ocamlbuild
```

4.1.2 Compiler avec make

Possibilité de compiler avec `make` seul, ou alors possibilité de rajouter des options. Cela va créer un fichier `main.native` sensé être executable sur n'importe quel système (nous n'avons testé que sur archlinux).

4.2 Utiliser directement l'executable

Il y a possibilité d'utiliser directement l'executable présent sur GitHub, pour cela il suffit de pipe l'instruction LaTeX à l'intérieur de `main.native` :

```
echo "3+2" | ./main.native
> 5
```

4.3 Interface graphique

— l'utilisateur peut rentrer une commande du format spécifié par l'interpréteur dans le champ de saisie principal
— en appuyant sur Maj+Entrée ou bien le bouton "Afficher l'expression" ou bien dans le menu "Affichage -> Afficher l'expression", l'utilisateur peut vérifier que ce qu'il a entré est du LaTeXcorrect.
— en appuyant sur Entrée ou bien le bouton "Afficher et calculer", l'utilisateur peut afficher le rendu LaTeXde son expression ainsi que le rendu LaTeXde la sortie du backend pour l'entrée donnée par l'utilisateur.
— en appuyant sur Ctrl+Shift+k ou bien le bouton "Effacer l'expression", ou bien dans le menu "Affichage -> Effacer l'expression", l'utilisateur peut effacer le contenu du champ d'entrée
— en appuyant sur Ctrl+Shift+e ou bien dans le menu "Affichage -> Effacer tout", l'utilisateur peut effacer le contenu du champ d'entrée ainsi que l'affichage LaTeX
— en appuyant sur Ctrl+Shift+c ou bien dans le menu "Fichier -> Remettre à zéro les variables", l'utilisateur peut faire oublier toutes les variables précédemment sauvegardées
— en allant dans le menu "Fichier -> Gérer les variables...", l'utilisateur peut afficher une fenêtre qui lui permettra de modifier ou bien de supprimer les variables déjà sauvegardées de manière sélectives (pas encore implémenté)
— en appuyant sur Ctrl+q ou bien dans le menu "Fichier -> Quitter", l'utilisateur peut fermer la fenêtre et quitter le logiciel

Chapitre 5

Conclusion

5.1 Interface graphique

5.1.1 Problèmes rencontrés

L'approche du c++ proposée à travers Qt est bien différente que celle qui a pu être proposée par la résolution de problèmes d'algorithmique lors d'un apprentissage sur openclassrooms ou codewars, notamment du fait que les classes de base du c++ sont supplantées par des classes propres à Qt, qui ont en conséquence aussi leurs méthodes qui leur sont propres.

De ce fait, trouver une documentation compréhensible sans base en c++ standard était compliqué, mais l'apprentissage du c++ ne permettait pas non plus toujours de m'éclairer sur des points d'ombre de la documentation Qt. Un exemple qui me revient à l'esprit serait le fait que les paramètres du moteur de rendu web de classe QWebEngineView est géré par une classe QWebEngineView ou encore la semaine que j'ai passé avant de trouver comment lier correctement un bouton de la barre de menu à une fonction.

C'est par ailleurs pour cette raison que j'ai préféré inclure les librairies MathJax directement dans le html passé à l'instance QWebEngineView plutôt que d'utiliser des méthodes de Qt que j'avais du mal à comprendre.

Aussi, il me semble que l'apprentissage de l'orienté objet en c++ dans le détail aurait pu grandement m'aider dans ma démarche, au vu de l'aide que l'apprentissage des bases m'a apportée dans la compréhension du fonctionnement global de Qt. Compréhension globale qui était nécessaire étant donné que les questions d'utilisateurs en ligne portaient souvent sur des exemples trop spécifiques pour être applicables à mon cas particulier.

Cependant, je (Léo Lagarrigue) n'ai aucun regret sur le fait d'avoir appris le c++ en autodidacte, en partie grâce à ce projet, c'est même le contraire.

5.1.2 Organisation

Une remarque que j'aurais est que le fait de ne pas avoir d'heures de travail dédiées dans l'emploi du temps hebdomadaire à ce projet du fait des disponibilités de chaque membre de l'équipe ne coïncidant pas nécessairement lors des weekends, jours fériés et autres jours de pause, n'aide pas à avancer sur le projet à un rythme régulier, ce qui aurait sans doute pu être préférable.

5.2 Interpréteur

5.2.1 Général

Ce projet un peu ambitieux au vu du temps disponible a été un succès partiel. L'objectif principal de faire un programme prenant en entrée une expression mathématique, la transformant en faisant des opérations sur des arbres et sortant un résultat correct, le tout en OCaml est rempli. Cependant, la

partie mathématique intéressante n'a malheureusement pas pu être abordée par manque de temps. Notamment l'objectif d'une calculatrice "formelle", à part une simplification partielle et la dérivation d'expression, nous n'avons pas réussi à faire grand chose. Cependant, ce projet était une excellent manière d'apprendre à travailler en groupe, d'essayer de surmonter les problèmes rencontrés quand on code à plusieurs sur un langage partiellement maîtrisé. Ce qui a été le plus intéressant est le fait d'avoir fait le plus de choses à la main et d'avoir joué avec la structure compliquée et rigoureuse demandée par le langage.

5.2.2 Les problèmes rencontrés

Les principaux problèmes sont liés au temps et au manque cruel de documentation. Nous avons été ambitieux, passer d'une maîtrise du OCaml écrit (en classe préparatoire on s'est concentré sur l'élaboration de courtes fonctions optimisées, le tout sur feuille, forcément plus facile pour la compilation :) à l'OCaml de production nécessite bien plus que cinquantes heures par personne. Nous n'avons pas voulu rentrer dans l'aspect trop technique du langage et nous avons donc utilisé le plus possible les outils que nous connaissions. Nous avons donc commencé le projet en gérant tout avec des types. Des modules aux erreurs, on a tout essayé de faire uniquement avec une bonne gestion de types. Ce qui n'a évidemment pas marché. Voici par exemple comment on a géré les exceptions avec des types :

```
variable.ml
let get key =
        try Some(Dict.find key !variables) with
            Not_found -> None
```

Ici, l'utilisation du type `option (type 'a option = Some of 'a | None)` permet de faire remonter des erreurs et de les traiter quand on le souhaite. C'est d'ailleurs la technique la plus utilisée dans la documentation que l'on a trouvé.
Le deuxième gros manque de documentation venait du module utilisé : `menhir`. Ne voulant pas passer trop de temps sur l'aspect technique de l'interprétation (on attend avec impatience le cours de compilation de deuxième année) nous avons été bloqué à plusieurs reprises.

5.2.3 Améliorations

C'est très frustrant de s'arrêter quand on commence à être à l'aise avec le langage et que le projet commence à prendre forme. Il y a plusieurs améliorations envisageables.

- Faire une refonte totale de la structure du projet, tout découper en classes/ interfaces pour que tout le projet puisse profiter de l'encapsulation.
- Modifier les types pour être plus précis et éviter le plus possible les redondances.
- Faire passer le projet sur `dune` au lieu de `ocamlbuild` et enfin avoir un système de build propre.
- Créer une syntaxe adaptée, le LaTeX était une bonne idée, mais ce n'est pas adapté au problème, il y a beaucoup trop d'implicite dans ce langage.
- Implémenter des vrais calculs formels intéressants.
- Créer un système d'exception plus adapté.
- Utiliser des functors pour profiter du langage.
- Utiliser les tests durant la création du projet et non à la fin.

Vive les chameaux

Bibliographie

[1] A. Bostan (2010) Journées Nationales de Calcul Formel

Annexe A

Gestion de projet

A.1 Plan de charge / Suivi d'activité

Sur le fichier xls

A.2 Planning prévisionnel

Pour le 5 Mars :
- Yasemine : Coder les premières fonctions de calcul (fonctions sur scalaires)
- Chabane/Clément/Thomas : Implémenter les premières fonctions de l'interpréteur / maîtriser la théorie
- Léo : Afficher du Latex avec GTK

Pour le 12 Mars :
- Yasemine : Implémenter la multiplication rapide de polynômes ainsi que le pgcd
- Chabane/Clément/Thomas : Proposer une première version de l'interpréteur
- Léo : Proposer une première version d'affichage en Latex depuis une chaîne de caractères normalisée

Pour le 16 Mars (réunion encadrant) :
- Proposer une première version du projet

Pour le 19 Mars :
- Yasemine : Proposer une première version qui communique avec les autres modules
- Chabane/Clément/Thomas : proposer une première version qui gère tous les modules
- Léo : Proposer une première version qui envoit/reçoit des données et les affiche

Jusqu'au 31 Mars : Résoudre les éventuels problèmes et remplir le rapport

Annexe B

Code source

Pour l'instant tout le code se trouve sur le GitHub du projet :

`https://github.com/Camelram-Beta/camelram-beta`

Envoyer un message à thomas.winninger@telecom-sudparis.eu pour devenir collaborateur.