



HAL
open science

Programmer en OCaml pour et sur les calculatrices Numworks

Laura Ly, Lilian Besson, Basile Pesin, Emmanuel Chailloux

► **To cite this version:**

Laura Ly, Lilian Besson, Basile Pesin, Emmanuel Chailloux. Programmer en OCaml pour et sur les calculatrices Numworks. JFLA 2026 – 37es Journées Francophones des Langages Applicatifs, Marie Kerjean; Yannick Zakowski, Jan 2026, Oberbronn, Alsace, France. hal-05428160

HAL Id: hal-05428160

<https://hal.science/hal-05428160v1>

Submitted on 21 Dec 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Programmer en OCaml pour et sur les calculatrices Numworks

Laura Ly¹, Lilian Besson²,
Basile Pesin³ et Emmanuel Chailloux⁴

¹Sorbonne Université, Paris, France

²Lycée Kléber, Strasbourg, France

³Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France

⁴Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Dans cet article, nous présentons le portage d’OMicroB, un environnement d’exécution pour le langage OCaml, sur les calculatrices graphiques programmables Numworks, aujourd’hui très présentes dans les collèges et lycées en France. Ce portage permet de programmer et compiler via un ordinateur des applications graphiques pour la calculatrice, comme un “jeu de la vie”, mais aussi d’exécuter des interpréteurs pour d’autres noyaux de langages de programmation (Prolog, mini-ML). Enfin, nous décrivons l’implémentation d’une boucle d’interaction (REPL) basée sur l’interpréteur du projet CamlBoot vue comme un environnement de développement permettant d’écrire et d’exécuter des petits programmes OCaml directement sur la calculatrice.

1 Introduction

Les premières expériences de programmation passent souvent par les calculatrices scientifiques du collège ou du lycée. Ce premier contact peut avoir une forte influence, en particulier sur la vision de ce qu’est la programmation, pour ces potentiels futurs programmeurs.

Les calculatrices graphiques récentes (Casio Graph 35+EII, Texas Instrument 83, Numworks N0120, ...) proposent de programmer directement sur la calculatrice dans des langages des familles BASIC, et plus récemment avec des portages du langage Python. Elles offrent de nombreuses modalités d’interaction : facilité de communication, affichage sur un écran graphique qui tient dans la main, lecture des entrées utilisateur via les différentes touches qui s’approchent d’un clavier d’ordinateur, et dans certains cas un système de fichiers local. Parmi ces calculatrices, la Numworks a l’avantage d’être livrée avec le système d’exploitation open-source Epsilon [Num], qui permet la programmation d’applications en C, C++, et Rust. Par ailleurs, la Numworks intègre nativement un éditeur et un environnement d’exécution pour le langage Python, dans sa version MicroPython [Tol17].

Dans cet article, nous montrons comment programmer la Numworks en OCaml [LDF⁺21] un langage multi-paradigme dont le noyau fonctionnel-impératif statiquement typé nous paraît un bon cadre pour de premiers pas en programmation, et qui est déjà utilisé dans l’enseignement supérieur en France.

Notre objectif est donc d’ajouter le support du langage OCaml pour la programmation pour et sur les calculatrices Numworks.

Bien que les calculatrices modernes deviennent de plus en plus performantes, leurs ressources restent néanmoins faibles par rapport à un ordinateur de bureau. La dernière version des calculatrices Numworks, la N0120, est équipée d'un processeur ARM Cortex-M7 à 550 Mhz, avec 564 Kio de mémoire vive (RAM), dont 148Kio sont accessibles aux applications externes, et 8 Mio de mémoire persistente (flash). Il est donc difficile de faire tenir dans ces espaces mémoire la distribution OCaml, même réduite à la machine virtuelle et au compilateur de byte-code.

Nous nous intéressons donc naturellement à OMicroB [VVC18], un environnement d'exécution pour OCaml adapté aux architectures à faibles ressources. OMicroB lit le byte-code d'un programme OCaml et l'embarque dans un programme C contenant un nouvel interprète de byte-code ainsi qu'une nouvelle bibliothèque d'exécution.

Notre première contribution est donc un portage d'OMicroB pour Numworks, ce qui fournit un chemin de construction d'exécutables pour la calculatrice. Nous avons aussi ajouté à la bibliothèque standard d'OMicroB des fonctions spécifiques à la Numworks, en particulier pour gérer les entrées/sorties. Cela permet de construire des applications interactives.

Dans un second temps, pour que les utilisateurs puissent programmer directement sur la calculatrice, nous ajoutons une boucle d'interaction (REPL pour Read-Eval-Print-Loop). Sachant que la boucle OCaml est trop volumineuse pour s'exécuter sur ce type de matériel, nous réutilisons en l'adaptant l'interprète développé pour le projet CamlBoot [CLS22]. Celui-ci, écrit en OCaml, permet d'évaluer tout code source OCaml à partir de l'arbre de syntaxe construit, avec seulement une vérification dynamique des types. Le but est alors de construire la boucle REPL à partir de cet interprète OCaml dans OMicroB pour en faire une application Numworks, et ainsi pouvoir écrire et tester ses développements sur la calculatrice en toute autonomie. Nous offrons ainsi une méthode de développement ludique et directe sur la calculatrice via cet interprète. Implémenter une REPL OCaml sur l'architecture de la Numworks permet d'attirer le programmeur en herbe, qui appréciera par la suite les garanties fournies par une compilation avec typage statique.

Ce court article détaille ces deux étapes (sections 2 et 3), en abordant les points techniques majeurs et en les illustrant par des tests et des applications s'exécutant sur la calculatrice Numworks. La conclusion (section 4) fait le point sur cette expérimentation, discute de son usage tout en tenant compte de ses limitations, et ouvre des voies pour de futurs travaux.

2 Portage d'OMicroB sur Numworks

OMicroB [VVC18] est un environnement d'exécution pour le langage de programmation OCaml [LDF⁺21] adapté aux micro-contrôleurs, des architectures à faibles ressources. OMicroB intègre une machine virtuelle (VM), implémentée en C pur et donc portable, pouvant exécuter le byte-code issu du compilateur OCaml.

Cela a permis de porter OMicroB sur plusieurs architectures : d'abord AVR, sur laquelle sont basées les cartes de prototypage Arduino, puis PIC32, permettant l'utilisation de matériel un peu plus puissant, et plus récemment Arm v7 [VPVC23], permettant l'exécution de programmes OCaml sur la carte pédagogique `micro:bit` [AP22] de la BBC.

2.1 Le projet OMicroB

Le schéma global de la chaîne de compilation d'OMicroB est donné en fig. 1. Le programme source est compilé de manière classique, en utilisant une bibliothèque standard spécifique au micro-contrôleur ciblé. Le fichier byte-code obtenu est passé à l'outil externe `ocamlclean` qui élimine le code mort et les déclarations globales inutiles. Le résultat est ensuite passé à `bc2c` (*byte-code to C*) qui embarque le byte-code dans un fichier C, sous forme de plusieurs tableaux représentant les instructions du programme, les variables globales, le tas et la pile de la machine virtuelle (VM).

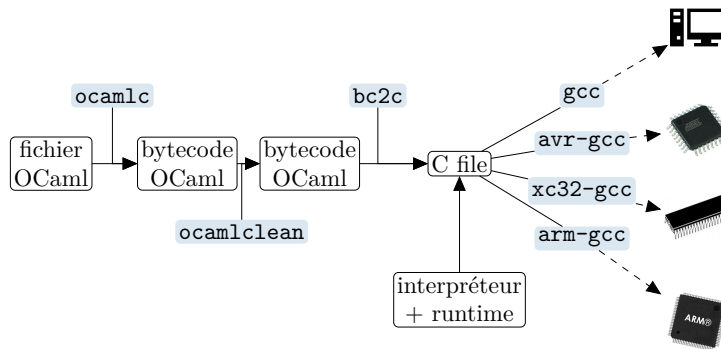


Figure 1. La chaîne de compilation avec OMicroB.

De nombreuses options permettent de minimiser les tailles mémoire. Les valeurs sont uniformes et représentées par un flottant par une technique de *NaN-boxing* [Nys21] en utilisant l'espace des NaN (*Not a Number*) pour coder les valeurs immédiates et les pointeurs du tas. Les instructions de la VM non utilisées dans le programme peuvent être éliminées de l'interprète. Deux algorithmes de ramasse-miettes (GC, *garbage collector*) sont fournis : un Stop&Copy et un Mark&Compact, privilégiant respectivement la rapidité d'exécution ou l'occupation mémoire. Un mécanisme d'évaluation partielle permet d'exécuter le programme à la compilation jusqu'à la première expression non évaluable. La mémoire flash peut être utilisée pour stocker des valeurs globales immuables comme décrit dans [VPVC23].

2.2 Portage et création d'exécutables Numworks via OMicroB

Grâce à l'architecture modulaire d'OMicroB, ajouter un portage pour nouvelle architecture est relativement facile. Pour le portage vers Numworks, nous avons ajouté un module au programme principal `omicrob` qui décrit les commandes à appliquer pour compiler le fichier C généré par `bc2c`. Nous avons aussi ajouté un fichier C contenant certaines fonctions attendues par l'interprète (fonctions à appeler à l'initialisation et à la fin du programme, en cas d'erreur, etc.). Enfin, nous avons ajouté un module `Numworks` à la bibliothèque standard d'OMicroB, qui contient des fonctions d'interaction avec le matériel. Celles-ci utilisent des fonctions `external` implémentées en C que nous avons également ajoutées.

La commande suivante permet de compiler `prog.ml` pour la Numworks avec des mots de 32 bits, une pile de 1024 mots et un tas de 2048 mots, en utilisant le GC Mark&Compact.

```
omicrob -device numworks -arch 32 -stack-size 1024 -heap-size 2048 -gc MAC \
  prog.ml -o prog.hex
```

Par défaut OMicroB effectue le maximum d'optimisations pour minimiser l'occupation de la RAM, et enchaîne toutes les étapes de compilation (voir fig. 1) qui peuvent être tracées (option `-v`) ou décomposées en l'enchaînement de commandes suivantes :

commande OMicroB	programme utilisé
<code>omicrob -device numworks -c prog.ml</code>	<code>ocamlc</code>
<code>omicrob -device numworks prog.cmo -o prog.byte</code>	<code>ocamlclean</code>
<code>omicrob -device numworks prog.byte -o prog.c</code>	<code>bc2c</code>
<code>omicrob -device numworks prog.c -o prog.hex</code>	<code>gcc</code>

Le fichier d'extension `.hex` peut alors être transféré sur la calculatrice soit en passant sur le site officiel de transfert d'application, soit directement avec la commande `omicrob -device numworks -flash prog.hex`.

2.3 Tests de performance (benchmarks)

Nous avons comparé la puissance de la calculatrice Numworks (dernier modèle N0120), équipée d’un processeur cadencé à 550 MHz, avec celle de la carte `micro:bit` de première génération, équipée d’un processeur ARM cadencé à 16 MHz. Pour cela, nous avons réexécuté les programmes utilisés pour comparer les performances d’OMicroB avec celles de μ Python sur `micro:bit`, dans [VPVC23]. Ces programmes sont représentatifs des fonctionnalités d’OCaml : ils contiennent des fermetures (`church`), des fonctions récursives (`fibonacci` et `takeuchi`), avec allocation mémoire (`takeushi`), des calculs en virgule flottante (`integr`) et des objets (`object`). Ces programmes incluent également des algorithmes un peu plus complexes : tri par arbre binaire de recherche (`treesort`) et recherche par backtracking d’une solution au problème des n-reines (`nqueens`). En raison de la taille très limitée de la mémoire de la carte `micro:bit`, nous n’avons pas pu exécuter d’exemples beaucoup plus conséquents. Pour OMicroB, nous avons utilisé les options de compilation `-stack-size 400 -heap-size 1000`, qui allouent une taille suffisante pour exécuter ces programmes. Les résultats sont présentés ci dessous, et sont comparés avec les exécutions des programmes sur un ordinateur standard (PC).

Les résultats sont cohérents : les temps de calculs sur `micro:bit` sont toujours entre 5000 et 6000 fois plus longs que sur PC. La seule surprise provient du temps de calcul pour le programme calculant des intégrales (`integr`), ce qui peut être expliqué par l’absence d’une unité de calcul en virgule flottante sur la carte `micro:bit`. La calculatrice Numworks est entre 50 et 70 fois plus rapide que la carte `micro:bit`. Comme le facteur de fréquence entre les deux processeurs est de 34, le gain supplémentaire provient en grande partie de l’organisation mémoire plus performante du processeur Cortex M7 STM32H725 qui équipe la Numworks. Pour le programme `integr`, l’écart est moindre, puisque le Cortex M7 possède effectivement une unité de calcul en virgule flottante double précision.

Exemple	sur PC (i7 @ 4.5GHz)	sur <code>micro:bit</code> v1		sur Numworks		
	Temps (s)	Temps (s)	/ PC	Temps (s)	/ PC	/ <code>micro:bit</code>
<code>church</code>	0.26	1302	5007	26	100	2%
<code>fibonacci</code>	0.28	1482	5293	31	110	2%
<code>takeuchi</code>	0.05	297	5940	4	80	1.3%
<code>integr</code>	3	22 282	7427	304	101	1.3%
<code>object</code>	0.22	1282	5827	17	77	1.3%
<code>treesort</code>	0.5	2590	5180	38	76	1.4%
<code>nqueens</code>	3.8	21 793	5735	323	85	1.5%

Le gain de puissance et la taille disponible de la mémoire RAM (148 Kio) permettent d’envisager de faire tourner sur Numworks des applications OCaml plus conséquentes.

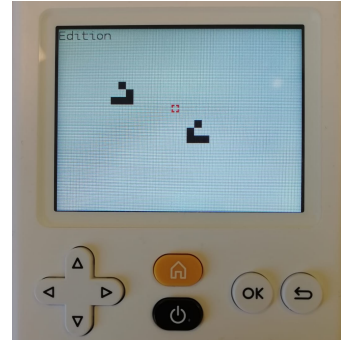
2.4 Applications interactives en OCaml sur calculatrices Numworks

La calculatrice Numworks propose de nouvelles modalités d’interaction aux programmes compilés avec OMicroB : un écran de 320 par 240 pixels et un clavier alpha-numérique. Pour les programmes C, l’accès à ces fonctionnalités est assuré par le kit de développement “EADK”, qui comprend une vingtaine de fonctions. Pour exposer ces fonctions d’interaction au niveau OCaml, nous avons implémenté un module `Numworks` dans la bibliothèque standard d’OMicroB. Un extrait de ce module est présenté dans la fig. 2a. Nous avons organisé les fonctions par sous-modules, chacun centré sur une fonctionnalité.

Le module `Color` permet la gestion des couleurs lors de l’affichage. Sur Numworks, les couleurs sont représentées par des entiers sur 16 bits, avec 5 bits pour les composantes rouge et bleue, et 6 bits pour la composante verte : c’est l’encodage dit “*high color*”. Le type `Color.t` encode une couleur ; la seule méthode permettant de construire une couleur est `Color.make`, qui reçoit les trois composantes rouge, verte et bleue, sous forme d’entiers.

```

module Color: sig
  type t
  val make: int -> int -> int -> t
end
module Screen: sig
  val fill_rect: Color.t -> int -> int -> int -> int -> unit
  val clear: unit -> unit
  val print: string -> int -> int -> unit
end
module Key: sig
  type t = Key_left | Key_up | ...
  val of_char: char -> t
  val to_char: t -> char
end
module Keyboard: sig
  val scan: unit -> unit
  val key_down: Key.t -> bool
  val wait_key_press: unit -> Key.t
end
    
```



(a) Extrait du module Numworks : fonctions d'interaction.

(b) Jeu de la vie sur Numworks.

Figure 2. Fonctions d'interactions et exemple d'usage pour une application interactive.

Le module `Screen` gère l'affichage sur l'écran de la Numworks : `fill_rect c x y w h` dessine un rectangle de couleur `c` à la position `(x, y)` sur l'écran (l'axe des abscisses est orienté de gauche à droite, et celui des ordonnées de haut en bas). La fonction `clear` réinitialise l'écran. Enfin, `print` affiche une chaîne de caractères à une position donnée. Nous avons également implémenté les fonctions OCaml usuelles d'affichage de valeurs (`print_string`, `print_int`, etc.), du module `Stdlib` d'OCaml, en gérant automatiquement la position d'écriture par un "curseur", modifié par effet de bord à chaque appel.

Les deux derniers modules permettent la gestion du clavier. Le premier, `Key`, contient une représentation des touches de la calculatrice comme un type énuméré, et des fonctions faisant la conversion avec les caractères correspondants. Le module `Keyboard` permet de lire l'état du clavier, avec deux approches différentes. Une première avec les fonctions `scan` et `key_down` qui fonctionnent en tandem : la première lit l'état du clavier et le stocke dans une variable interne, et la deuxième indique si une touche donnée est pressée. Une seconde avec la fonction `wait_key_press` qui bloque jusqu'à ce qu'une touche soit pressée puis la renvoie. L'implémentation C de cette fonction utilise un appel système qui évite l'attente active.

Application : jeu de la vie sur Numworks Pour démontrer les possibilités d'interaction disponibles sur Numworks, nous avons implémenté en OCaml le "Jeu de la vie" [Gar70], un automate cellulaire proposé par J. Conway. Le monde est représenté par un tableau de booléens de taille 32 par 24, encapsulé dans un objet `world`, qui expose en particulier des méthodes `nextGen` et `draw`. En mode édition, l'état initial du monde peut être modifié en changeant chaque cellule. Appuyer sur EXE lance la simulation. Appuyer sur la touche "retour" ↵ réinitialise le monde à son état initial et retourne en mode édition. L'application est implémentée en une centaine de lignes d'OCaml, qui font appel aux fonctions présentées plus haut pour interagir avec l'utilisatrice de la Numworks (voir fig. 2b). L'application résultante utilise environ 60Kio de mémoire flash et nécessite au minimum une pile de 120 mots et un tas de 4000 mots, soit environ 16Kio de mémoire RAM au total.

2.5 Lecture de fichiers et exécution d'interpréteurs

La calculatrice Numworks propose un éditeur initialement conçu pour les programmes Python. Il est aussi possible d'éditer, sauvegarder et synchroniser ces fichiers depuis le site de

Numworks (<https://my.numworks.com/python>). Bien que ce système ait été conçu pour Python, il peut être utilisé pour écrire ou transférer des fichiers textes quelconques, qui pourront ensuite être lus par nos applications : par exemple, les fichiers sources pour les interpréteurs mini-Prolog et mini-ML présentés plus loin.

Les fichiers créés sont sauvegardés dans la mémoire ROM de la calculatrice, linéairement entre deux adresses connues. Les données pour chaque fichier suivent un format simple : longueur du fichier sur deux octets, suivi de son nom terminé par un caractère nul, suivi d'un octet de méta-données utilisées par l'interpréteur Python, suivi du contenu du fichier. Nous avons adapté une bibliothèque libre, *extappStorage* [Cou], permettant de lire ces fichiers locaux créés sur la calculatrice.

Au niveau OCaml, nous avons implémenté les fonctions usuelles du module `Stdlib`. La fonction `open_in : string -> in_channel` ouvre le fichier indiqué ; le type abstrait `in_channel` correspond à une valeur sur 64 bits qui encode un pointeur sur le prochain caractère à lire, et le nombre de caractères restants dans le fichier. Nous avons également défini les fonctions `input` et `input_char`, qui lisent depuis ce canal en mettant à jour le pointeur et la taille restante. Disposer de ces fonctions de la bibliothèque standard simplifie le portage d'applications existantes, comme pour les deux exemples suivants.

Un interprète pour un mini-Prolog. Nous avons pu porter sur Numworks un interprète pour un mini-Prolog, écrit en quatre cents lignes d'OCaml pur, qui avait été initialement développé à but pédagogique. Cet interprète non-interactif lit une théorie Prolog basique et des questions depuis deux fichiers `prolog_theory` et `prolog_questions`, puis répond aux questions en cherchant des preuves dans la théorie en utilisant un algorithme standard de recherche par *backtracking*.

Un interprète pour un mini-ML. Nous avons également porté sur Numworks un interprète pour un langage jouet "à la ML", implémenté en cinq cents lignes d'OCaml et du code généré `ocamllex` et `ocamlyacc`, qui avait été initialement développé par un groupe d'étudiants de niveau L3 pour un projet de cours de compilation. L'analyseur lexical produit par `ocamllex` et l'analyseur syntaxique produit par `ocamlyacc` ont été compilés en sources OCaml puis intégrés dans le programme. L'interprète lit un programme source depuis un fichier `minicaml`, puis l'exécute en évaluant son arbre de syntaxe abstraite.

3 Une boucle d'exécution pour OCaml sur Numworks

Maintenant que des programmes OCaml conséquents et interactifs s'exécutent sur la calculatrice Numworks, nous cherchons à pouvoir la programmer directement en OCaml via une boucle REPL. À la différence d'implantations du langage Scheme, en particulier le système Ribbit [YF21] dont la boucle REPL tient sur 4Ko, la boucle OCaml est gourmande en ressources dans la mesure où il est nécessaire d'embarquer les analyseurs lexical et syntaxique, le typeur, et le compilateur de byte-code ; tout cela nécessite d'importantes capacités mémoire que la calculatrice ne possède pas.

3.1 Approches envisagées

Plusieurs approches ont été explorées afin de permettre l'interprétation de code OCaml sur l'environnement fortement contraint de la calculatrice Numworks. Elles diffèrent selon leur niveau d'intégration avec la chaîne de compilation OCaml existante et leur compatibilité avec les contraintes matérielles de la plateforme cible.

Exécuter `ocamlc` sur Numworks Une première idée consistait à embarquer directement le compilateur `ocamlc`, qui est implémenté en OCaml, sur la calculatrice. En théorie, le

code source écrit sur calculatrice pourrait être passé au compilateur, puis le bytecode généré pourrait être exécuté par OMicroB. C'est analogue au fonctionnement du toplevel `ocaml` pour PC, qui appelle la machine virtuelle `ocamlrun`. En pratique, ce n'est pas faisable puisque (1) OMicroB lit le bytecode embarqué dans du code C et pas dans le format sérialisé produit par `ocamlc` et (2) la représentation des valeurs d'OMicroB diffère de celle d'`ocamlc`. Il faudrait donc embarquer, en plus du compilateur `ocamlc` (dont les analyseurs lexical et syntaxique, le typeur et le générateur de byte-code), `bc2c` et stocker le byte-code engendré en Ram, ce qui va augmenter les ressources mémoire utilisées.

Transpilation vers JavaScript avec Js_of_OCaml et Espruino Une variante de l'approche ci-dessus serait d'exécuter `ocamlc` puis de compiler le bytecode OCaml en JavaScript, via le compilateur `js_of_ocaml` [VB14] qu'il faudrait aussi embarquer sur la calculatrice. Nous pourrions ensuite exécuter le résultat dans un moteur JavaScript léger pour microcontrôleur, comme celui du projet Espruino [Ltd]. Pour explorer cette piste, nous avons porté l'interprète Espruino comme une application Numworks implémentée en C. Cette approche n'a pas été explorée davantage, mais elle se révélerait sans doute trop lourde pour la Numworks : la charge mémoire d'un programme OCaml compilé vers JavaScript et interprété par Espruino dépasserait rapidement les limites de l'appareil.

Exécuter l'interprète de CamlBoot dans OMicroB La piste la plus prometteuse repose ainsi sur la combinaison de Camlboot [CLS22] et OMicroB. Camlboot propose un interprète OCaml minimaliste, écrit en OCaml lui-même, conçu pour reconstruire la chaîne de compilation dans une logique de *bootstrapping*, ce qui permettrait d'utiliser bien moins de mémoire sur la calculatrice que les solutions précédemment envisagées. En l'exécutant sur la machine virtuelle d'OMicroB, il devient envisageable d'obtenir un environnement OCaml interactif embarqué, formant la base d'une future boucle REPL minimale pour Numworks.

3.2 Portage de l'interprète CamlBoot

Notre portage cible la version 4.09 de CamlBoot [CLS]. Cette version a l'avantage d'utiliser le moteur d'exécution du générateur d'analyseurs syntaxiques Menhir [PRG21] écrit intégralement en OCaml, et donc immédiatement portable dans OMicroB, contrairement au moteur d'exécution d'`ocamlyacc`, dont nous aurions dû réimplémenter les composantes C. Pour pouvoir exécuter CamlBoot via OMicroB, la première étape a été d'isoler les modules nécessaires à son fonctionnement. CamlBoot utilise le dépôt officiel du compilateur OCaml, dont nous avons extrait les modules utiles : `Parsetree` et ses dépendances, `Lexer` et `Parser`. Pour ces derniers, nous avons directement copié les fichiers `.ml(i)` générés par `ocamllex` et `menhir`, ce qui simplifie la chaîne de compilation de notre application.

Ces composants font appel à des modules de la bibliothèque standard d'OCaml qui n'étaient pas, jusqu'ici, supportés par OMicroB, avec en particulier le module `Lazy` pour traiter les flots de lexèmes de manière paresseuse durant l'analyse syntaxique. Dans l'implémentation standard d'OCaml, le type `'a Lazy.t` repose sur une représentation interne spécifique et sur des primitives écrites en C (via `CamlinternalLazy`), qui exploitent des invariants mémoire incompatibles avec la machine virtuelle légère d'OMicroB. Pour lever cette incompatibilité, nous avons réimplémenté en OCaml le type et ces primitives. Pour cela, nous utilisons une référence mutable associée à un type algébrique qui représente l'état de la valeur (non évaluée, en cours d'évaluation ou déjà calculée). Cette réécriture préserve la sémantique observable de `Lazy` dans le contexte du parseur, tout en restant compatible avec l'environnement.

Nous avons également importé les modules spécifiques à CamlBoot (`Eval` et ses dépendances). Ces modules dépendent également de fonctions de la bibliothèque standard non disponibles sur OMicroB (dont le module `Format`). Nous avons donc modifié le code de CamlBoot pour supprimer les références à ces fonctions, qui n'étaient heureusement utiles qu'au débogage. Nous avons complété ce portage avec une boucle d'exécution interactive

(REPL), implémentée en utilisant les fonctions présentées dans la section précédente. Avant de commencer, la boucle charge le code de la bibliothèque standard qui est embarqué sous forme de chaîne de caractères dans le programme. Ensuite, la boucle permet de charger des fichiers créés avec l'éditeur de la calculatrice (directive `%use`), et d'évaluer des définitions/expressions écrites directement dans la boucle. Chaque expression peut dépendre de toutes les définitions précédentes, y compris celles chargées depuis un fichier. L'application développée utilise environ 330Kio de mémoire flash. Un exemple de son utilisation est présenté en fig. 3.

```

Camlboot for Numworks 1.0
%use file.py;; to load a file
> %use ocaml.py;;
()#0
(144, "fibo 12", 144)
> fibo;;
<function>
> fibo 13,,
233
> "Hello JFLA 2026 !";;
"Hello JFLA 2026 !"
    
```

Figure 3. Application REPL basée sur Camlboot sur Numworks.

3.3 Performances et usage de la mémoire

Pour évaluer la performance de notre portage de CamlBoot, nous réutilisons les benchmarks présentée plus haut. Nous comparons trois exécutions du même programme sur la calculatrice Numworks : le programme compilé avec OMicroB (colonne centrale), comme dans la table précédente, le programme s'exécutant dans l'interprète CamlBoot (colonne de droite), et un programme Python équivalent s'exécutant dans l'interprète Python fourni avec la calculatrice. Ce dernier nous sert de référence. Par ailleurs, nous mesurons aussi les tailles de piles et de tas minimum, ainsi que la quantité de RAM correspondante, pour pouvoir exécuter chaque programme dans CamlBoot. Les résultats sont présentés dans la table ci-dessous.

Exemple	μ Python		OMicroB (compilé)		OMicroB + CamlBoot				
	Taille (loc)	Temps (s)	Temps (s)	/ μ Py	Temps (s)	/ μ Py	Pile (mots)	Tas (mots)	RAM (Kio)
church	15	144	26	18%	8628	60	300	12 000	49
fibonacci	14	186	31	16%	14 000	75	350	11 500	47
takeuchi	14	76	4	5%	1251	16	300	12 000	49
integr	26	51	31	60%	2627	51	1000	35 000	144
object	16	69	17	24%	6811	98	300	14 000	57
treeseort	51	41	18	44%	6603	161	2000	35 000	148
nqueens	52	2577	323	12%	156 530	60	2100	34 500	146

Les programmes compilés avec OMicroB sont entre 2 et 20 fois plus rapides que les programmes Python équivalents. Le calcul d'intégrales (`integr`) effectuant principalement des calculs sur les flottants obtient un facteur d'efficacité de 2, par contre la version de `takeushi` manipulant des triplets alloue beaucoup et gagne un facteur 20. En revanche, notre portage de CamlBoot est beaucoup plus lent (entre 15 et 100 fois) que l'interprète MicroPython. En effet, ce dernier est implémenté directement en C, et compile le programme source en bytecode avant son exécution, ce qui est plus efficace que d'évaluer l'arbre de syntaxe directement comme le fait CamlBoot.

Par ailleurs, pour les plus "gros" programmes (n-reines, calcul d'intégrales et tri), la quantité de mémoire vive utilisée s'approche dangereusement des 148Kio de RAM disponible

pour les applications externes. En effet, CamlBoot doit stocker en mémoire l’arbre de syntaxe du programme à exécuter, mais aussi les valeurs intermédiaires calculées durant son évaluation : `nqueens` conserve les solutions trouvées avant d’en retourner le nombre, `integr` s’approche de la solution flottante par des appels récursifs successifs, et `treessort` construit des arbres. Pour réduire la taille de l’arbre de syntaxe, il y a plusieurs solutions :

- supprimer les informations inutiles à CamlBoot (en particulier les annotations de position dans le source). Cela nécessiterait de modifier le type de données représentant l’arbre de syntaxe, ce qui rendrait une future évolution vers une nouvelle version d’OCaml plus complexe.
- stocker l’arbre de syntaxe de la bibliothèque standard dans la mémoire flash, beaucoup plus grande que la RAM : pour cela, on pourrait écrire l’arbre de syntaxe “en dur” dans le code source de l’application, plutôt que de le faire construire dynamiquement par l’analyseur syntaxique de CamlBoot.

Cependant, ces solutions n’auront pas d’impact sur la mémoire utilisée par les valeurs construites par CamlBoot. En particulier, une fermeture est allouée pour chaque fonction définie dans le programme : cela limite la taille possible de la bibliothèque standard incluse avec notre distribution (pour le moment, seul le module `Stdlib` est disponible).

Ces contraintes sur la mémoire ne permettent pas d’envisager d’ajouter un typeur, en particulier le typeur d’OCaml, dans cette REPL pour Numworks. Il serait certes envisageable de l’extraire, en limitant certaines dépendances, pour que le code tienne en mémoire flash, mais les besoins mémoire (tas et pile) du typage ne tiendraient pas dans ce si petit espace.

4 Conclusion

Nous avons présenté le portage d’OMicroB sur la calculatrice Numworks, permettant d’exécuter des programmes OCaml utilisant tous les traits du langage. Notre développement¹ est disponible sur <https://github.com/stevenvar/OMicroB/tree/numworks>.

Parmi les programmes supportés par cet environnement, le plus ambitieux est un portage de l’interprète du projet CamlBoot, sous la forme d’une boucle d’interaction REPL permettant de charger des fichiers sources OCaml écrits dans l’éditeur de la calculatrice. Cela constitue un environnement de prototypage pratique : les fichiers sources écrits et testés directement sur la calculatrice peuvent ensuite être exportés sur PC, et recompilés pour PC ou pour Numworks via OMicroB. Ainsi, même si notre boucle REPL ne dispose pas de typage statique, puisque CamlBoot évalue l’arbre de syntaxe non typé, le programme peut toujours être vérifié sur PC après son export. Utiliser l’interprète de CamlBoot sur une calculatrice à la mémoire limitée est un tour de force. Ces limitations mémoires impactent la portion de la bibliothèque standard OCaml offerte dans notre REPL. Comme discuté plus haut, il y a plusieurs voies pour dépasser ces limitations.

Pour étendre notre travail, nous aimerions proposer un portage de la bibliothèque `Graphics` de OCaml, afin de faciliter le portage d’applications graphiques existantes, et l’apprentissage de la programmation graphique en OCaml, directement sur une calculatrice scientifique.

Nous envisageons aussi d’expérimenter l’usage de la Numworks avec OCaml dans un contexte pédagogique réel, une classe de CPCG en filière MP2I, pour que les élèves qui y découvrent la programmation avancée via le langage OCaml puissent programmer avec leur calculatrice Numworks. Ces calculatrices étant très présentes au collège et lycée, nous envisageons d’explorer davantage leur utilisation dans un contexte éducatif, en particulier pour l’enseignement de la programmation.

Remerciements Nous remercions Benoît Vaugon et Loïc Sylvestre pour l’aide et les discussions sur ce travail ainsi que les relecteurs anonymes pour leurs retours encourageants.

1. Les travaux sur OMicroB sont soutenus par l’Initiative de Recherche et Innovation sur le Logiciel Libre (IRILL).

Références

- [AP22] Sandeep Saini ASHWIN PAJANKAR, Abhishek Sharma : *BBC Micro:bit in Practice*. Packt Publishing, 2022.
- [CLS] Nathanaëlle COURANT, Julien LEPILLER et Gabriel SCHERER : Camlboot 4.09. <https://github.com/gasche/camlboot/tree/4.09>.
- [CLS22] Nathanaëlle COURANT, Julien LEPILLER et Gabriel SCHERER : Debootstrapping without Archeology - Stacked Implementations in Camlboot. *Art Sci. Eng. Program.*, 6(3), 2022.
- [Cou] Yaya COUT : Numworks extapp storage. <https://framagit.org/Yaya.Cout/numworks-extapp-storage>.
- [Gar70] Martin GARDNER : Mathematical games. *Scientific American*, 223(4), 1970.
- [LDF⁺21] Xavier LEROY, Damien DOLIGEZ, Alain FRISCH, Jacques GARRIGUE, Didier RÉMY et Jérôme VOUILLON : The OCaml system (release 4.13). <https://ocaml.org/manual/4.13/index.html>, septembre 2021.
- [Ltd] Pur3 LTD : Espruino - javascript for microcontrollers. <https://www.espruino.com/>.
- [Num] NUMWORKS : Epsilon operating system. <https://github.com/numworks/epsilon>.
- [Nys21] Robert NYSTROM : *Crafting Interpreters*. Genever Benning, 2021.
- [PRG21] François POTTIER et Yann RÉGIS-GIANAS : The menhir parser generator. <https://gitlab.inria.fr/fpottier/menhir/>, 2005–2021.
- [Tol17] Nicholas H TOLLERVEY : *Programming with MicroPython : embedded programming with microcontrollers and Python*. ” O’Reilly Media, Inc.”, 2017.
- [VB14] Jérôme VOUILLON et Vincent BALAT : From bytecode to javascript : the js_of_ocaml compiler. *Softw. Pract. Exper.*, 44(8), août 2014.
- [VPVC23] Steven VAROUMAS, Basile PESIN, Benoît VAUGON et Emmanuel CHAILLOUX : Programming microcontrollers through high-level abstractions : The OMicroB project. *Journal of Computer Languages*, 77, 2023.
- [VVC18] Steven VAROUMAS, Benoît VAUGON et Emmanuel CHAILLOUX : A Generic Virtual Machine Approach for Programming Microcontrollers : the OMicroB Project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS’18)*, 2018.
- [YF21] Samuel YVON et Marc FEELEY : A small scheme vm, compiler, and repl in 4k. In *Workshop on Virtual Machines and Intermediate Languages (VMIL@SPLASH’21)*, 2021.