

Conception et implémentation d'un compilateur pour le langage E

1 But de ces séances de travaux pratiques

Au cours de ces 5 séances de travaux pratiques, vous allez réaliser un compilateur pour un petit langage de programmation, le langage E. Le langage E est un dérivé du langage C. Nous nous concentrerons d'abord sur la compilation d'une version basique du langage E que nous enrichirons par la suite sur le temps additionnel. Votre compilateur sera composé d'un analyseur lexical (*lexer*), d'un analyseur syntaxique (*parser*), puis de plusieurs passes de compilation qui transformeront les programmes E dans des langages de plus en plus bas niveau, jusqu'à la génération de code assembleur x86-32 et RISC-V, qui seront finalement assemblés par des assembleurs existants et qui pourront être exécutés sur vos machines.

Comme nous allons le voir, le langage E est relativement petit, pour vous permettre de le réaliser dans le temps de TP qui vous est imparti. Cependant, il permet d'illustrer un grand nombre de concepts fondamentaux de la compilation. Au cas où vous trouveriez que le langage est trop petit, ou bien que les passes de compilation et d'optimisation suggérées ne sont pas suffisantes, nous vous fournirons une liste d'améliorations possibles que vous pourrez implémenter.

La Section 2 vous présente l'architecture du compilateur que vous allez concevoir, notamment les structures de données à utiliser et les différents langages intermédiaires. La Section 3 vous présente l'infrastructure de test qui vous accompagnera pour déboguer votre compilateur. Les sections suivantes décrivent le travail que vous aurez à faire lors des séances de TP. Le découpage en TP est donné à titre indicatif. Si vous n'avez pas fini le travail demandé à la fin d'un TP, vous pourrez utiliser un bout de la séance suivante (ou de vos soirées) pour le finir. Essayez de ne pas prendre *trop* de retard.

Vous trouverez le squelette associé à ce TP à l'adresse suivante :

<https://gitlab-research.centralesupelec.fr/cidre-public/compilation/infosec-ecomp>

Si vous voulez travailler sur ce projet dans un dépôt git pour partager votre code avec votre binôme, utilisez la procédure suivante :

```
$ git remote rename origin le-remote-d-origine
```

2 Organisation du compilateur

La figure 1 donne un aperçu de la structure du compilateur que vous allez réaliser. À partir d'un fichier source *.e*, l'analyseur lexical (ou *lexer*) générera un flux de lexèmes (ou *tokens*). Ce flux sera donné à l'analyseur syntaxique (ou *parser*) qui devra générer un arbre de syntaxe abstraite (*Abstract Syntax Tree*, ou AST). L'AST sera transformé en un programme E, puis en un programme CFG (*Control-Flow Graph*), ensuite un programme RTL et finalement un programme Assembleur. Chacun de ces langages intermédiaire est détaillé ci-dessous, et est illustré sur l'exemple de la Figure 2a.

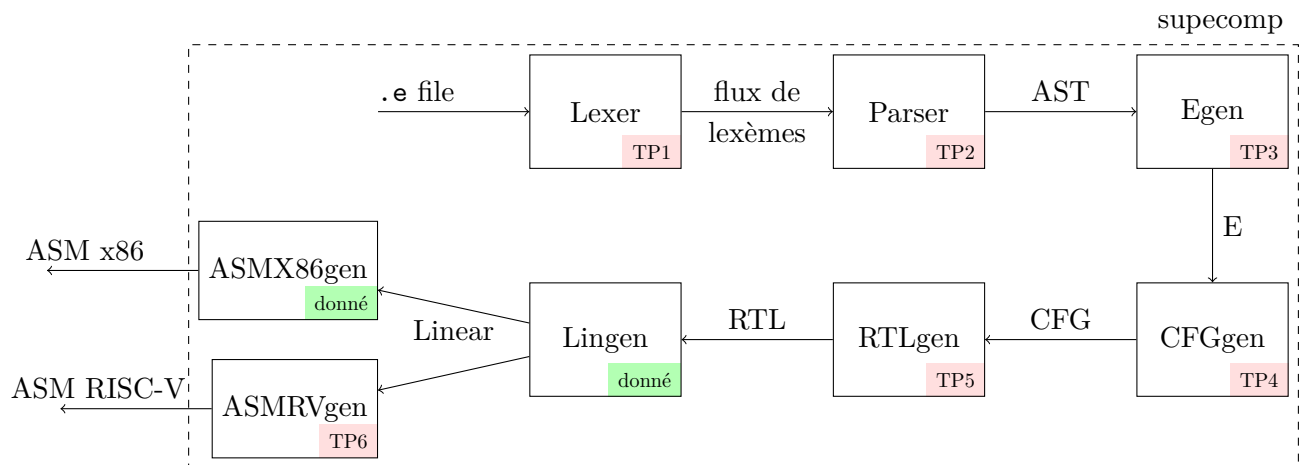


FIGURE 1 – Aperçu de la structure du compilateur

3 Tests

Vous trouverez dans le répertoire **tests** un ensemble d'outils vous permettant de tester votre compilateur. Les dossiers **array**, **basic**, **funcall**, **real_args**, **strings** et **struct** contiennent des programmes E vous permettant de tester les fonctionnalités correspondant au nom du dossier. Cependant durant les séances de TPs nous nous concentrerons essentiellement sur les tests du dossier **basic**.

Pour chaque fichier **test.e**, nous vous avons fourni la sortie attendue avec les paramètres 1, 2 et 3 dans **test.e.expect_1_2_3** et avec les paramètres 14, 12, 3, 8 et 12 dans **test.e.expect_14_12_3_8_12**. Vous pouvez tester que votre compilateur est conforme à ce qui est attendu en lançant **make test** depuis la racine de votre projet.

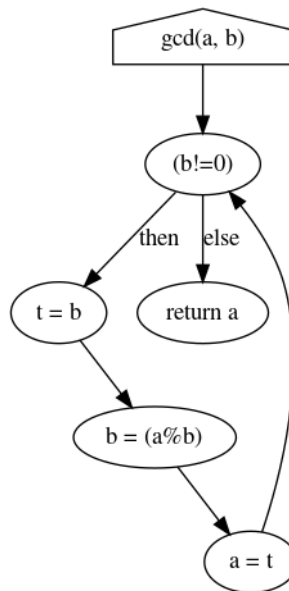
Les résultats des tests seront rassemblés dans le fichier **tests/results.html** que vous pouvez visualiser avec votre navigateur préféré. Les résultats sont présentés sous forme de tableaux où les lignes correspondent aux programmes testés et les colonnes les résultats obtenus à différentes étapes de la compilation. Le nom des programmes sont cliquables pour avoir plus d'informations sur le déroulement de sa compilation.

Pour tester les programmes individuellement vous pouvez utiliser le script **tests/test.py** ou directement le binaire produit (**main.native**) avec **make** :

```
# Commandes utiles
$ tests/test.py -f tests/basic/toto.e
$ tests/test.py --help
$ ./main.native --help
```

```
gcd(a,b){
  while(b != 0){
    t = b;
    b = a % b;
    a = t;
  }
  print a;
  return a;
}
```

(a) Programme E



(b) CFG correspondant

```
gcd(r0, r1):
goto n2
n2:
r3 <- 0
r2 <- r1 <> r3
r2 ? goto n5 : goto n1
n5:
r4 <- r1
goto n4
n4:
r5 <- r0 % r1
r1 <- r5
goto n3
n3:
r0 <- r4
goto n2
n1:
return r0
```

(c) Programme RTL

```
.global supecomp_main
supecomp_main:
push ebp
mov ebp, esp
sub esp, 20
.n2:
mov DWORD PTR [ebp+-12], 0
xor ebx, ebx
mov ecx, DWORD PTR [ebp+12]
mov edx, DWORD PTR [ebp+-12]
cmp ecx, edx

setne bl
mov DWORD PTR [ebp+-8], ebx
mov eax, DWORD PTR [ebp+-8]
test eax, eax
jnz .n5
jmp .n1
.n1:
mov eax, DWORD PTR [ebp+8]
jmp .ret
```

```
.n5:
mov ebx, DWORD PTR [ebp+12]
mov DWORD PTR [ebp+-16], ebx
jmp .n4
.n4:
mov eax, DWORD PTR [ebp+8]
mov ebx, DWORD PTR [ebp+12]
mov edx, 0
div ebx
mov DWORD PTR [ebp+-4], edx
mov ebx, DWORD PTR [ebp+-4]
mov DWORD PTR [ebp+12], ebx
jmp .n3
.n3:
mov ebx, DWORD PTR [ebp+-16]
mov DWORD PTR [ebp+8], ebx
jmp .n2
.ret: leave
ret
```

(d) Assembleur x86-32

FIGURE 2 – Les différents langages intermédiaires utilisés lors de la compilation d'un programme

4 TP1 : Analyseur lexical

Le but de cette séance de TP est de réaliser un analyseur lexical pour le langage E. Cette séance est l'occasion de mettre en œuvre l'algorithme vu en cours pour la réalisation d'un analyseur lexical. On vous rappelle qu'il repose sur l'utilisation d'un automate déterministe à états finis. Afin d'accélérer votre développement nous allons vous fournir une partie du code, et vous proposer une organisation de votre code.

4.1 Fonctions utiles de la librairie standard OCaml

La documentation complète est disponible en ligne sur : <https://caml.inria.fr/pub/docs/manual-ocaml/libref>

- `List.mem (e: 'a) (l: 'a list): bool`
retourne vrai si `e` est dans la liste `l`
- `List.fold_left (f: 'a -> 'b -> 'a) (acc: 'a) (l: 'b list): 'a`
applique `f` à chaque élément de `l` en stockant le résultat dans `acc`
- `List.map (f: 'a -> 'b) (l: 'a list): 'b list`
retourne une liste dans laquelle `f` a été appliquée à chaque élément de `l`.
- `List.filter_map (f: 'a -> 'b option) (l: 'a list): 'b list`
Comme `List.map`. De plus, les éléments pour lesquels `f` retourne `None` sont retirés de la liste retournée.
- `Set` les mêmes fonctions existent pour les ensembles `Set`
- `Set.union (s1: Set.t) (s2: Set.t): Set.t`
retourne l'union des ensembles `s1` et `s2`
- `Set.add (e: elt) (s: Set.t): Set.t`
ajoute l'élément `e` à l'ensemble `s`
- `Hashtbl.find_opt (tbl: ('a, 'b) Hashtbl.t) (k: 'a) -> 'b option`
retourne l'élément associé à la clé `k`. Si cette clé n'existe pas dans `tbl` retourne `None`.
- `Hashtbl.find_replace (tbl: ('a, 'b) Hashtbl.t) (k: 'a) (v: 'b) : unit`
associe la valeur `v` à la clé `k` dans la table `tbl`.

4.2 Tests

Pour compiler et tester votre code il suffit de lancer `make test` dans le répertoire racine de votre projet. Les résultats des tests sont stockés dans le fichier `tests/results.html`.

Résultats attendus sur un exemple :

```
$ cat tests/basic/just_a_variable_37.e
main(){
    just_a_variable = 37;
    return just_a_variable;
}
```

```
# Au début du TP dans results.html:
Lexing error:
Lexer failed to recognize string starting with
↳ 'main(){
    just_a_var'
```

```
# À la fin du TP, results.html:
SYM_IDENTIFIER(main)
SYM_LPARENTHESIS
SYM_RPARENTHESIS
SYM_LBRACE
SYM_IDENTIFIER(just_a_variable)
SYM_ASSIGN
SYM_INTEGER(37)
SYM_SEMICOLON
SYM_RETURN
SYM_IDENTIFIER(just_a_variable)
SYM_SEMICOLON
SYM_RBRACE
SYM_EOF
```

Lorsque vous appelez `make test`, votre compilateur est lancé sur 30 fichiers de tests (les fichiers `tests/basic/*.e`). Pour le moment, le fichier `tests/results.html` indique que tous ces tests échouent à l'analyse lexicale puisque votre analyseur n'est pas encore écrit. Au fur et à mesure des séances de TP, ce fichier vous donnera de plus en plus d'information, notamment le résultat de l'analyse lexicale, syntaxique ainsi que le résultat de l'exécution de chacun des programmes de test à différents niveaux dans la chaîne de compilation. Cela sera un bon moyen de valider la correction de vos passes de compilation.

Vous pouvez aussi lancer le compilateur « à la main », c'est-à-dire sans passer par le `make test` :

```
$ make
$ ./main.native -f tests/basic/just_a_variable_37.e -show-tokens -
```

pour lancer le compilateur sur le fichier `tests/basic/just_a_variable_37.e` et afficher les tokens reconnus. (Le « - » à la fin de la ligne de commande indique qu'on souhaite afficher les tokens sur la sortie standard. Si on veut les écrire dans un fichier, on remplacera ce « - » par le nom du fichier.)

4.3 Débogage

Pour vous aider à déboguer, les fonctions suivantes sont à votre disposition :

- `nfa_to_string` et `dfa_to_string` transforment des `nfa` et `dfa` en chaînes de caractères (`string`)
- `nfa_to_dot` et `dfa_to_dot` construisent une représentation visuelle d'un `dfa` dans un fichier `*.dot`. Le fichier `*.dot` pourra ensuite être transformé en image via la commande `$ dot fichier.dot -Tsvg -o fichier.svg`.

Vous trouverez des exemples d'utilisations de ces fonctions dans le fichier `src/test_lexer.ml`. Ce fichier contient ce qui s'apparente à une fonction `main` en C : une déclaration de fonction `let () = ...`. Dans cette fonction, une liste d'expressions régulières est créée, affichée, transformée en NFA. Ce NFA est affiché avec `nfa_to_string` d'une part et `nfa_to_dot` d'autre part, ce qui génère un fichier `/tmp/nfa.dot`, que vous pouvez convertir en image SVG avec la commande suivante :

```
$ dot -Tsvg /tmp/nfa.dot -o /tmp/nfa.svg
```

Ou bien en utilisant un convertisseur en ligne, ici par exemple : <https://dreampuf.github.io/GraphvizOnline>. Le NFA est ensuite déterminisé, le DFA résultant est affiché puis écrit dans

/tmp/dfa.dot.

Pour lancer ces tests, il vous suffit de lancer la commande `make test_lexer` depuis le répertoire `src`.

4.4 Travail à effectuer

Le développement de notre analyseur lexical se déroule en trois étapes. Premièrement, la spécification des expressions régulières permettant de reconnaître les termes du langage E. Ensuite, un NFA (*Non-deterministic Finite Automaton*) pourra être généré pour ces expressions régulières. Finalement, ce NFA sera transformé en DFA (*Deterministic Finite Automaton*) qui sera utile à l'analyseur pour reconnaître les termes du langage E et les associer au bon lexème. Le travail que vous réaliserez au cours de cette séance se déroulera dans les fichiers `src/lexer_generator.ml` et `src/e_regexp.ml`.

4.4.1 Expressions régulières du langage E

Un premier travail est de donner à l'analyseur différentes expressions régulières permettant d'identifier les mots-clés et noms de variables du langage E. Pour vous familiariser avec le langage E, n'hésitez pas à parcourir le répertoire `tests/basic`, où une trentaine d'exemples vous sont donnés.

Question 4.1. Compléter la fonction `list_regexp` du fichier `src/e_regexp.ml` en remplaçant les regex `Eps` par une expression régulière adéquate. À noter que les variantes `regex` sont définies plus haut dans le fichier et que la liste de symboles est disponible dans le fichier `src/symbols.ml`.

4.4.2 Expressions régulières en NFAs

Nous souhaitons maintenant produire le NFA correspondant aux expressions régulières utilisées pour analyser le langage.

Question 4.2. Écrire les fonctions `cat_nfa`, `alt_nfa` et `star_nfa` du fichier `src/lexer_generator.ml`. Ces fonctions permettent respectivement la concaténation, l'union et la répétition d'automates `nfa`.
Le type `nfa` est décrit et commenté au début du fichier.

Question 4.3. Compléter la fonction `nfa_of_regexp` qui produit un NFA à partir d'une expression régulière. Les cas `Eps` et `Charset c` sont donnés en exemple. Traiter les variantes restantes de `regexp`.

4.4.3 Détermination d'un NFA en DFA

Cette partie se charge de transformer un NFA en DFA en suivant les étapes décrites en cours. Le type `dfa` utilisé dans cette partie est défini et commenté dans le fichier `src/lexer_generator.ml`.

Question 4.4. Compléter les fonctions `epsilon_closure` et `epsilon_closure_set` qui retournent les états accessibles par ε -transitions d'un état ou d'un ensemble d'états d'un graphe `nfa`. Plus particulièrement, il faut écrire la fonction récursive `traversal` pour `epsilon_closure`.

Question 4.5. Construire l'expression `transitions` de la fonction `build_dfa_table`. Cette fonction permet de construire la table de transitions d'un `dfa` à partir d'un `nfa`. Les différentes étapes permettant de construire cette table de transitions sont spécifiées dans les commentaires situées au-dessus de la fonction.

Question 4.6. Compléter les fonctions `min_priority` et `dfa_final_states` permettant de définir les états finaux de notre `dfa`. Les états finaux d'un `dfa` correspondent aux états qui contiennent au moins un état final d'un `nfa`. La fonction de conversion associée à un état final est obtenue en faisant usage de `min_priority`.

Question 4.7. Pour finir la construction de notre DFA, compléter la fonction de transition `make_dfa_step` en utilisant la table de transition construite précédemment avec `build_dfa_table`.

4.4.4 Obtention de lexèmes à l'aide du DFA

Nous avons maintenant obtenu un DFA capable de reconnaître les mots-clés et variables du langage E. Nous souhaitons maintenant que notre DFA décompose les chaînes de caractères d'un programme E en une série de lexèmes/jetons définis dans le fichier `src/symbols.ml`.

Question 4.8. Complétez la fonction `tokenize_one`. Celle-ci contient une fonction récursive `recognize` qui effectue des transitions dans le DFA tant que possible et retourne un jeton lorsqu'il aboutit.

*Note : vous aurez besoin de la fonction `string_of_char_list` qui transforme un **char list** en **string***

Nous vous offrons les dernières étapes, à savoir

- la fonction `tokenize_all` qui répète `tokenize_one` tant qu'il y a des lexèmes à lire,
- la fonction `tokenize_file` qui transforme un nom de fichier en la liste des lexèmes qui sont reconnus dans ce fichier,
- les morceaux de code dans `main.ml` qui appellent le lexer.

Si tout va bien, un `make test` maintenant vous affiche plein d'erreurs, mais de syntaxe seulement :-)

A Installation des dépendances

Dès le début, vous aurez besoin d'OCAML.

```
# Install opam
$ sudo apt install opam # ou avec votre gestionnaire de paquets favori
$ opam init
$ opam switch install 4.08.0 (ou plus)
# Installation des dépendances
$ opam install stdlib-shims ocamlbuild ocamlfind menhir lwt logs batteries yojson websocket
↪ websocket-lwt-unix
$ eval $(opam env)
```

Pour le TP6, vous aurez besoin d'outils spécifiques :

- `riscv64-unknown-elf-gcc` : un *cross-compileur* pour RISC-V pour générer des exécutables RISC-V,
- `qemu-riscv64` : un émulateur de RISC-V pour les exécuter.

Sur des Debian (10+) ou Ubuntu (19.04+) :

```
$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu
↪ binutils-riscv64-linux-gnu
```

Sur ArchLinux :

```
$ sudo pacman -S riscv64-linux-gnu-binutils riscv64-linux-gnu-gcc riscv64-linux-gnu-gdb
↪ qemu-arch-extra
```

Si vous devez compiler vous-mêmes : (attention c'est long!)

```
# GCC:
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
# install dependencies
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk
↪ build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
# configure and build (vous pouvez changer le préfixe, c'est ici que seront installés les outils)
$ cd riscv-gnu-toolchain
$ ./configure --prefix=/usr/local
$ sudo make

# QEMU:
$ wget https://download.qemu.org/qemu-4.1.0.tar.xz
$ tar xf qemu-4.1.0.tar.xz
$ cd qemu-4.1.0
$ ./configure --disable-kvm --disable-werror --prefix=/usr/local --target-list="riscv64-softmmu"
$ make
$ sudo make install
```


B Merlin l'assistant de programmation OCaml

Merlin est un assistant à la programmation OCaml pouvant s'interfacer avec de nombreux éditeurs de code tels que Emacs, Vim, Atom, Visual Studio Code, Sublime Text ... Vous pouvez trouver le dépôt Github de Merlin à l'url suivante : <https://github.com/ocaml/merlin>.

B.1 Installation de Merlin

Avec Opam :

```
opam install merlin
opam user-setup install # configure Emacs et Vim pour Merlin
```

Manuellement : Instructions et sources sur le dépôt Github

B.2 Merlin et Visual Studio Code

Les instructions sont tirés du blog suivant : <https://www.cosmiccode.blog/blog/vscode-for-ocaml/>

1.

```
opam install merlin
opam install ocp-ident # Outil formatant le code OCaml
```
2. Dans VSCode chercher et installer l'extension *OCaml and Reason IDE*
3. Si VSCode n'arrive pas à trouver les outils de l'environnement OCaml-Merlin il est possible de les spécifier de la manière suivante
 - **Ctrl+P > Workspace Settings**
 - Extensions → Reason configuration
 - Spécifier les chemins vers les binaires `ocamlmerlin`, `ocamlfind`, `ocp-ident` et `opam`