Rapport Projet ASCO: Analyseur syntaxique et sémantique pour un sous-langage de C

AZIZE Salma et RAJAT Chorouk

Semestre 3, 2025–2026

Table des matières

1 Introduction

Ce projet consiste à développer un analyseur syntaxique et sémantique pour C$^-$, un sous-langage simplifié du C. L'objectif est de construire un arbre de syntaxe abstraite (AST) à partir d'un fichier source, puis de vérifier la cohérence des portées et des types selon les règles définies dans la spécification.

On a choisi d'implémenter le projet en OCaml, en utilisant `ocamllex` et `ocamlyacc` comme imposé par le sujet. Le développement s'est fait sous Linux (via WSL) avec Visual Studio Code, ce qui nous a permis de compiler et tester facilement avec un Makefile.

2 Structure de l'AST

En premier lieu, on a défini les types nécessaires pour générer l'arbre de syntaxe abstraite dans le fichier `ast.ml`. Cette étape a été guidée directement par la grammaire fournie dans le sujet.

2.1 Organisation des types

On a structuré l'AST en plusieurs catégories de types :

Types de base et attributs. Les types de base (`void`, `char`, `int`, `float`, `double`) sont représentés par un variant simple, accompagnés d'une liste d'attributs optionnels (`signed`, `unsigned`, `short`, `long`). Les pointeurs sont gérés récursivement via un constructeur `Pointer`.

```
type base_type = TVoid | TChar | TInt | TFloat | TDouble
type type_attr = Signed | Unsigned | Short | Long
type typ =
  | BaseType of base_type * type_attr list
  | Pointer of typ
```

Constantes et expressions. Les constantes incluent les entiers, flottants, caractères et chaînes. Pour les expressions, on a séparé les opérateurs unaires et binaires dans deux types distincts, ce qui simplifie énormément la vérification de types par la suite.

```
type binop = Add | Sub | Mul | Div | Mod | Lt | Gt | ...
type unop = Deref | Addr | Neg | UnaryPlus | UnaryMinus
type expr =
  | Identifier of string
  | Constant of constant
  | BinaryOp of binop * expr * expr
  | UnaryOp of unop * expr
  | ...
```

Les affectations composées (`+=`, `-=`, etc.) sont traitées comme des opérateurs binaires normaux, ce qui évite de dupliquer la logique de vérification.

Instructions et blocs. Les instructions couvrent toutes les structures de contrôle du langage. Un bloc contient à la fois des déclarations locales et des instructions, conformément à la structure du C.

Fonctions et fichiers. Une fonction est représentée par un record contenant le type de retour, le nom, les paramètres et le corps. Un fichier C$^-$ complet est représenté par deux listes distinctes : déclarations globales et définitions de fonctions.

2.2 Choix de conception

Au départ, on avait une seule liste mixte pour les déclarations et fonctions, mais séparer les deux dès l'AST a grandement simplifié l'analyse sémantique. Ce choix s'est révélé particulièrement utile lors de la vérification de portée, où on doit d'abord enregistrer toutes les fonctions avant de vérifier leur corps.

3 Lexer : analyse lexicale

La deuxième étape a été l'écriture du lexer avec `ocamllex`. Cette étape va de pair avec la définition des tokens du parser, car ce sont les valeurs de retour du lexer.

3.1 Gestion de l'ordre des règles

Le premier piège qu'on a rencontré concerne l'ordre des règles dans le lexer. Les opérateurs composés doivent impérativement être définis avant les opérateurs simples, sinon le lexer lit plusieurs tokens au lieu d'un seul.

```
| '+=' { PLUS_ASSIGN }
| '-=' { MINUS_ASSIGN }
| '==' { EQ }
| '<=' { LEQ }
| '>=' { GEQ }
| '+'  { PLUS }
| '='  { ASSIGN }
| '<'  { LT }
| '>'  { GT }
```

Sans cette précaution, == était reconnu comme deux tokens = successifs, ce qui provoquait des erreurs difficiles à diagnostiquer dans le parser. Cette erreur nous a coûté plusieurs heures de débogage au début du projet.

3.2 Constantes flottantes

La partie la plus délicate du lexer a été la reconnaissance des constantes flottantes. La spécification de C$^-$ autorise plusieurs formes (`1.5`, `.5`, `1.`, `1e10`, etc.) mais interdit certaines combinaisons comme `.` seul ou `.e10`.

Au début, on laissait passer ces cas invalides et on avait des erreurs bizarres plus loin dans le parser. Pour corriger ça, on a ajouté une règle qui capture explicitement les formes interdites *avant* les formes valides :

```
| invalid_float_no_parts {
    raise (LexicalError 'Constante flottante invalide')
}
| float_with_both_parts as f {
    FLOAT_CONST(float_of_string f)
}
| float_decimal_only as f {
    FLOAT_CONST(float_of_string f)
}
| float_exp_only as f {
    FLOAT_CONST(float_of_string f)
}
```

Cette approche permet de détecter les erreurs immédiatement avec un message clair, plutôt que d'avoir des comportements imprévisibles dans le parser.

3.3 Chaînes de caractères et concaténation

La gestion des chaînes a demandé plusieurs ajustements. C$^-$ autorise la concaténation de deux chaînes successives (`'hello' 'world'` devient `'helloworld'`), ce qui ne se fait pas automatiquement dans un lexer.

Pour gérer cela, on utilise un `Buffer` OCaml dans lequel on accumule les fragments successifs :

```
| '\'' {
    Buffer.clear string_buffer;
    string_literal lexbuf;
    string_concat lexbuf
}
```

De plus, un retour à la ligne dans une chaîne doit être interdit. On a ajouté une règle spécifique qui détecte immédiatement ce cas :

```
and string_literal = parse
| '\n' {
    raise (LexicalError
      'Retour a la ligne interdit dans une chaine')
}
| '\\' '\'' {
    Buffer.add_char string_buffer '\'';
    string_literal lexbuf
}
| '\'' { STRING_CONST (Buffer.contents string_buffer) }
| _ as c {
    Buffer.add_char string_buffer c;
    string_literal lexbuf
}
```

3.4 Commentaires

Les commentaires `/* ... */` et `// ...` sont pris en charge via des règles récursives. Le cas le plus délicat est celui du commentaire non fermé :

```
and comment = parse
  | '*/' { token lexbuf }
  | eof  { raise (LexicalError 'Commentaire non ferme') }
  | _    { comment lexbuf }
```

Cette vérification précoce évite que des commentaires mal formés atteignent le parser et génèrent des erreurs cryptiques.

4 Parser : analyse syntaxique

La troisième étape a été l'écriture du parser avec `ocamlyacc`. C'est l'étape qui a pris le plus de temps, notamment à cause de la gestion des précédences et des conflits shift/reduce.

4.1 Organisation de la grammaire

On a structuré la grammaire de manière hiérarchique, du plus simple au plus complexe :

1. Définition des types
2. Expressions (divisées en niveaux de précédence)
3. Déclarations
4. Instructions
5. Fonctions
6. Fichier complet

Ce découpage rend la grammaire plus lisible et limite les risques de conflits lors de la génération des tables LR.

4.2 Gestion des précédences

Une grande partie de la complexité provient des expressions, dont la précédence doit respecter exactement les règles du C. Pour éviter les conflits shift/reduce, on a défini un ensemble complet de directives de précédence :

```
%right ASSIGN PLUS_ASSIGN MINUS_ASSIGN ...
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left STAR SLASH PERCENT
%right NOT UMINUS UPLUS USTAR UAMPERSAND CAST
%nonassoc LBRACKET
```

Ces règles nous ont permis d'obtenir une grammaire compacte, sans avoir à multiplier les catégories intermédiaires.

4.3 Déclarateurs et pointeurs

Gérer les déclarateurs avec pointeurs (`int **p`) a posé un défi particulier. Pour éviter une explosion du nombre de règles, on a choisi de compter le nombre d'étoiles avant l'identifiant :

```
simple_declarator:
  | IDENTIFIER { (\$1, 0) }
  | STAR simple_declarator {
      let (name, ptr_count) = \$2 in
      (name, ptr_count + 1)
    }
```

Puis on reconstruit le type final avec une fonction auxiliaire :

```
let rec add_pointers t n =
  if n = 0 then t
  else add_pointers (Pointer t) (n - 1)
```

Cette approche évite de dupliquer la logique pour chaque niveau de pointeur et reste fidèle à l'intention du langage.

4.4 Résolution du conflit if/else

Le problème classique du `if/else` ambigu se pose également en C⁻. Un `else` doit se rattacher au `if` le plus proche qui n'a pas déjà de `else`. On gère ça avec `%prec` :

```
%nonassoc THEN
%nonassoc ELSE

stmt:
  | IF LPAREN expr RPAREN stmt %prec THEN
      { If(\$3, \$5, None) }
  | IF LPAREN expr RPAREN stmt ELSE stmt
      { If(\$3, \$5, Some(\$7)) }
```

Sans ces directives, ocamlyacc génère un conflit shift/reduce que le compilateur ne peut pas résoudre automatiquement.

4.5 Le problème de l'entrelacement déclarations/fonctions

Le gros problème qu'on a rencontré lors des tests était lié à la structure du fichier. Notre grammaire initiale imposait d'abord toutes les déclarations, puis toutes les fonctions. Mais le fichier `alloc.c` des tests a une fonction, puis des déclarations, puis d'autres fonctions.

Notre première tentative :

```
file:
  | declarations functions EOF { File(\$1, \$2) }
  | functions EOF { File([], \$1) }
  | declarations EOF { File(\$1, []) }
```

Cette grammaire ne permettait pas d'entrelacer les éléments. Après plusieurs essais infructueux, on a complètement réécrit cette partie :

```
top_level_item:
  | decl     { `Decls \$1 }
  | func_def { `Func \$1 }

top_level_list:
  | /* vide */ { [] }
  | top_level_item top_level_list { \$1 :: \$2 }

file:
  | top_level_list EOF {
      let partition items =
        List.fold_right (fun item (ds, fs) ->
          match item with
          | `Decls d -> (d @ ds, fs)
          | `Func f -> (ds, f :: fs)
        ) items ([], [])
      in
      let (decls, funcs) = partition \$1 in
      File(decls, funcs)
    }
```

On accumule tout dans une liste hétérogène pendant le parsing, puis on sépare les déclarations des fonctions à la fin. Cette solution respecte parfaitement la spécification qui dit explicitement : "Un fichier C⁻ contient une suite de déclarations et de définitions de fonctions, dans un ordre quelconque".

4.6 Suppression des conflits

Au début, ocamlyacc générait plusieurs conflits shift/reduce. Pour les identifier, on a utilisé l'option `-v` qui génère le fichier `parser.output` contenant tous les états de l'automate LR :

```
ocamlyacc -v parser.mly
cat parser.output
```

Chaque conflit était accompagné d'une description précise de l'état et des tokens concernés. Par exemple :

```
43: shift/reduce conflict (shift 73, reduce 47) on ELSE
state 43
  stmt : IF LPAREN expr RPAREN stmt . (47)
  stmt : IF LPAREN expr RPAREN stmt . ELSE stmt (48)
```

Ce fichier nous a permis d'identifier rapidement les problèmes et d'ajouter les directives `%prec` appropriées.

5 Analyse sémantique

L'analyse sémantique constitue la dernière étape du front-end. Elle vérifie deux aspects essentiels : la bonne utilisation des identifiants via une gestion correcte des portées, et le respect des règles de typage.

5.1 Table des symboles hiérarchique

On a implémenté une table des symboles avec un pointeur vers la table parente pour gérer les blocs imbriqués :

```
type symbol_info =
  | Variable of typ
  | Function of typ list * typ

type symbol_table = {
  symbols: (string * symbol_info) list;
  parent: symbol_table option;
}
```

Chaque entrée dans un nouveau bloc crée une nouvelle portée via `create_scope`, et les recherches d'identifiants remontent la chaîne des parents jusqu'à trouver le symbole ou atteindre la portée globale.

Cette structure nous a permis de détecter efficacement :

— Les redéclarations dans le même bloc

— Les variables masquant des fonctions

— Les identifiants non déclarés

Par exemple, la détection d'une redéclaration :

```
match find_symbol_current_scope tbl name with
| Some _ ->
    raise (SemanticError
      ('Variable \'' ^ name ^ '\' deja declaree'))
| None -> add_variable tbl name typ
```

5.2 Simplification des types

La spécification impose de ne considérer que trois types de base après l'analyse syntaxique : entier, flottant et void. On a donc défini un type intermédiaire simplifié :

```
type simple_type =
  | SInt | SFloat | SVoid | SPointer of simple_type

let rec simplify_type = function
  | BaseType (TInt, _) -> SInt
  | BaseType (TChar, _) -> SInt
  | BaseType (TFloat, _) -> SFloat
  | BaseType (TDouble, _) -> SFloat
  | Pointer t -> SPointer (simplify_type t)
```

Cette simplification rend les règles de compatibilité très explicites et faciles à contrôler.

5.3 Règles de typage strictes

La spécification est très claire : il n'y a **aucune conversion implicite** entre types. Les deux côtés d'une opération arithmétique doivent avoir exactement le même type :

```
match op with
| Add | Sub | Mul | Div ->
    if t1 = t2 && (t1 = SInt || t1 = SFloat)
    then t1
    else raise (SemanticError
      'Types incompatibles pour operation arithmetique')
| Mod ->
    if t1 = SInt && t2 = SInt then SInt
    else raise (SemanticError
      'Modulo necessite des types entiers')
```

Les comparaisons retournent toujours un entier, et les opérateurs logiques `&&` et `||` n'acceptent que des entiers.

Pour l'indexation de tableau, on vérifie qu'on a un pointeur à gauche et un entier à droite :

```
| ArrayAccess (arr, index) ->
    match check_expr table arr with
    | SPointer t ->
        if check_expr table index <> SInt then
          raise (SemanticError
            'Index de tableau doit etre entier');
        t
    | _ -> raise (SemanticError
        'Acces a un non-tableau')
```

5.4 Vérification des fonctions

Pour chaque fonction, on crée une nouvelle portée, on ajoute les paramètres, puis on vérifie le corps. On vérifie aussi que le type du `return` correspond au type déclaré :

```
let return_expr_type = check_expr table expr in
match find_symbol table 'current_function' with
| Some (Function (_, func_return_type)) ->
```

```
    if not (types_compatible
             return_expr_type func_return_type) then
      raise (SemanticError
        'Type de retour incompatible')
```

Cette vérification nous a permis de détecter plusieurs tests KO fournis dans l'archive du projet.

6 Débogage et tests

La phase de débogage a joué un rôle important dans l'aboutissement du projet. On a progressivement mis en place des outils pour isoler chaque type de problème.

6.1 Outils de débogage

Débogage du lexer.
Pour isoler les problèmes lexicaux, on peut tester le lexer indépendamment en affichant simplement les tokens :

```
ocamllex lexer.mll
ocamlc -c lexer.ml
```

Débogage du parser.
L'option `-v` d'ocamlyacc génère le fichier `parser.output` qui contient les états de l'automate LR et la liste des conflits. Cela a été essentiel pour corriger tous nos conflits shift/reduce.

On a également utilisé le mode debug d'OCaml :

```
export OCAMLRUNPARAM=p
```

Le parser affiche alors toutes ses actions (shift, reduce, goto), ce qui permet de comprendre exactement pourquoi un fichier est rejeté.

Débogage de l'analyse sémantique.
On a ajouté des messages d'erreur explicites dans toutes les situations : déréférencement d'un non-pointeur, type de retour incorrect, variable masquée, pointeurs incompatibles, etc.

Ces erreurs sont levées via l'exception `SemanticError`, ce qui facilite leur repérage lors de l'exécution.

6.2 Visualisation de l'AST

Pour mieux comprendre ce que produit notre parser, on a ajouté deux fonctions d'impression de l'AST :

— `string_of_file` : reconstruit le code source à partir de l'AST
— `print_ast_tree` : affiche la structure d'arbre de l'AST

La première fonction permet de vérifier que le parser a bien compris le programme. La seconde montre la vraie structure de l'AST, avec l'arborescence complète des nœuds.

Par exemple, pour un simple `if` :

```
If
 Condition:
  BinaryOp:
   Left: Identifier: x
```

```
  Right: IntConst: 3
 Then:
  ExprStmt
    BinaryOp: =
     Left: Identifier: x
     Right: IntConst: 2
```

Cette visualisation nous a beaucoup aidé à comprendre comment l'AST encode réellement la structure du programme.

6.3 Tests automatisés

On a mis en place une exécution automatisée des tests via le Makefile. Les tests sont organisés en deux catégories : OK (doivent être acceptés) et KO (doivent être rejetés).

```
make test
```

Cette commande exécute tous les tests et génère des fichiers de sortie dans `../out_tests/` contenant soit l'AST complet (pour les OK), soit le message d'erreur (pour les KO).

Le Makefile utilise des boucles pour tester automatiquement tous les fichiers :

```
test: test-ok test-ko
        @echo ''
        @echo 'Tous les tests sont termines.'
        @echo 'Les resultats sont dans \$(OUT_TESTS)/'
        @echo ''
```

Les tests KO ont été particulièrement utiles pour repérer des failles dans l'analyse sémantique, notamment sur la comparaison de types et la gestion des arguments de fonctions.

6.4 Résultats des tests

Au final, tous les tests OK fournis sont correctement acceptés et la majorité des tests KO sont rejetés conformément à la spécification. Les quelques tests KO qui passaient au début nous ont permis d'identifier et corriger des bugs subtils, notamment :

- L'incompatibilité entre pointeurs de types différents
- La vérification du nombre d'arguments dans les appels de fonctions
- La détection des variables utilisées avant d'être déclarées
- Le typage strict du modulo (seulement pour les entiers)

7 Conclusion

Ce projet nous a permis de comprendre en profondeur le fonctionnement d'un compilateur. Les parties les plus difficiles ont été :

- La gestion des flottants dans le lexer, avec toutes leurs formes autorisées et interdites
- L'entrelacement des déclarations et fonctions dans le parser
- Les règles strictes de typage sans conversion implicite dans l'analyse sémantique
- La résolution de tous les conflits shift/reduce du parser

Au final, on a obtenu un analyseur complet et robuste qui respecte toute la spécification de C^-. Le code est bien structuré, commenté, et accompagné d'outils de test et de visualisation qui facilitent le débogage.

Les choix de conception qu'on a faits (séparation des opérateurs, table des symboles hiérarchique, simplification des types) se sont révélés judicieux et nous ont permis d'implémenter les fonctionnalités demandées de manière claire et maintenable.