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

Assembleur – Compilation, ensIIE

Semestre 3, 2025–26

1 Informations pratiques

Le code rendu comportera un Makefile, et devra pouvoir être compilé avec la commande `make`. **Tout projet ne compilant pas se verra attribuer un 0** : mieux vaut rendre un code incomplet mais qui compile, qu'un code ne compilant pas. Votre code devra être **abondamment commenté et documenté**.

L'analyseur syntaxique demandé à la question 2 sera impérativement obtenu avec les outils lex/yacc [1] ou ocamllex/ocamlyacc [2]. Ceci implique donc que votre projet sera écrit au choix en C ou en OCaml.

Des fichiers d'entrée pour tester votre code seront disponibles dans une archive à l'adresse : `http://web4.ensiie.fr/~guillaume.burel/compilation/projet_cmoins.tar.gz` . Vous attacherez un soin particulier à ce que ces exemples fonctionnent.

Votre projet est à déposer sur `http://exam.ensiie.fr` dans le dépôt `asco23-proj` sur forme d'une archive `tar.gz` **avant le 1er décembre 2025**. **Tout projet rendu en retard se verra attribuer la note 0.** Vous n'oublierez pas d'inclure dans votre dépôt un rapport (PDF) précisant vos choix, les problèmes techniques qui se posent et les solutions trouvées.

Le projet est à réaliser **en binôme**. Toute tentative de fraude (plagiat, etc.) sera sanctionnée. Si plusieurs projets ont **des sources trop similaires** (y compris sur une partie du code uniquement), *tous* leurs auteurs se verront attribuer la note 0/20.

1. Documentation disponible à la page `http://dinosaur.compilertools.net/`
2. " `https://ocaml.org/manual/lexyacc.html`

2 Sujet

Le but de ce projet est d'écrire un programme qui prend en entrée un programme écrit en C$^-$, un sous-langage de C, et qui construit son arbre de syntaxe abstraite. La syntaxe de C$^-$ est décrite par ce qui suit :

Les types seront les types de base `void char int float` et `double`, optionnellement préfixés par un ou plusieurs attributs parmi `signed unsigned short` et `long`, et optionnellement suffixés par une ou plusieurs `*` (pointeurs).

Les constantes seront :

— des constantes entières : un signe éventuel `+` ou `-` suivi soit d'une suite non vide de chiffres entre `0` et `9` qui ne commence pas `0` (notation décimale), soit d'un `0` suivi d'une suite de chiffres entre `0` et `7` (notation octale), soit de `0x` ou `0X` suivi de chiffres entre `0` et `9` et de lettres entre `a` et `f` (potentiellement en capitale) (notation hexadécimale) ;

— des constantes flottantes : un signe éventuel `+` ou `-` suivi d'une partie entière constituée de chiffres de `0` à `9`, d'un point, puis d'une partie décimale constituée de chiffres de `0` à `9`, puis d'un exposant formé par la lettre `e` ou `E` suivi d'un signe éventuel suivi d'une suite non vide de chiffres entre `0` et `9`. La partie entière ou la partie décimale peuvent éventuellement être vide, mais pas les deux à la fois. Le point ou la partie exposant peuvent être absents, mais pas les deux à la fois ;

— des constantes de chaînes de caractères : une suite de caractères entre deux `"` où on peut faire apparaître le caractère `"` à condition qu'il soit préfixé de `\`, et sans retour à la ligne. Deux constantes chaînes de caractères consécutives sont concaténées en une seule.

Les identifiants sont des suites non vides de lettres, de chiffres et d'underscore qui commencent par une lettre ou un underscore.

Une expression peut être :

— soit un identifiant ;

— soit une constante ;

— soit un appel de fonction : un identifiant suivi d'une liste potentiellement vide d'expressions séparées par des virgules, entre parenthèses (id`(`e`, `e`, ..., `e`)`) ;

— soit un accès dans un tableau : une expression suivie d'une autre entre crochets (e`[`e`]`), prioritaire sur ce qui suit ;

— soit le mot clef `sizeof` suivi d'un type entre parenthèses (`sizeof (`t`)`) ;

— soit un déréférencement : une `*` suivi d'une expression (`*`e) ;

— soit une adresse : un `&` suivi d'une expression (`&`e) ;

— soit une conversion explicite : un type entre parenthèses suivi d'une expression (`(`t`)` e) ;

— soit une négation `!`e, prioritaire sur ce qui suit ;

— soit une opération arithmétique binaire e *op* e où *op* est un de `+ - / * %`, avec les priorités usuelles, parenthésées à gauche ;

— soit une comparaison e *op* e où *op* est un de `< > <= >= == !=` , moins prioritaires que les opérations arithmétiques, parenthésées à gauche ;

— soit une opération logique binaire e *op* e où *op* est un de `&& ||`, moins prioritaires que les comparaisons, `&&` prioritaire sur `||`, parenthésées à gauche ;

— soit une affectation e *op* e où *op* est un de `= += -= /= *= %=`, moins prioritaires que les opérations logiques, parenthésées à droite ;

— soit une expression entre parenthèses (`(`e`)`) .

Une déclaration est un type suivi d'une liste d'identifiants séparés par des virgules, puis un point-virgule (t id`,` id`, ...,` id`;`).

Une instruction peut être :

— soit une expression suivi d'un point-virgule (e`;`) ;

— soit l'instruction vide représentée par un point-virgule seul ;

— soit un bloc qui commence par une suite de déclaration, puis une suite d'instructions, le tout entre accolades (`{` d `...` d i `...` i `}`) ; les suites peuvent être vides ;

— soit un retour : le mot-clef `return` suivi d'une expression et d'un point-virgule (`return` e`;`) ;

— soit une conditionnelle : le mot clef `if`, une expression entre parenthèses, une instruction, le mot-clef `else` et une instruction (`if (`e`)` i `else` i) ; la partie `else` i est optionnelle ; elle se rattache au `if` le plus proche qui n'a pas déjà de `else` ;

— soit une boucle while : le mot-clef `while` suivi d'une expression entre parenthèses et d'une instruction (`while (`e`)` i)

— soit une boucle do while : le mot-clef `do` suivi d'une instruction, d'une expression entre parenthèses et un point-virgule (`do` i `while (`e`);`) ;

— soit une boucle for : le mot-clef `for` suivi, entre parenthèses, de trois expressions optionnelles séparées par deux point-virgules, puis une instruction (`for (`e?`;` e?`;` e?`)` i) .

Une définition de fonction est un type de retour suivi d'un identifiant suivi, entre parenthèses, d'une liste potentiellement vide de successions d'un type et d'un identifiant, séparées par des virgules, le tout suivi d'un bloc (t id`(`t id`,` t id`, ...,` t id`)` `{` d `...` d i `...` i `}`).

Un fichier C$^-$ contient une suite de déclarations (variables globales) et de définitions de fonctions, dans un ordre quelconque.

On ignorera les lignes commençant par `#`. On utilisera les commentaires de C (`/* dans le commentaire /* toujours dedans */ en dehors`) et de C++ (`// jusqu'à la fin de la ligne`).

3 Questions

1. Définir des types de données correspondant à la syntaxe abstraite des programmes C$^-$ (couvrant toute la grammaire). En particulier on définira entre autres des

types `file`, `instruction` et `expression`. On pourra par exemple considérer que les boucles for et do while sont du sucre syntaxique pour des boucles while, de même pour les affectations `+= -= *= /= %=`.

2. À l'aide de lex/yacc, ou ocamllex/ocamlyacc, écrire un analyseur lexical et syntaxique qui lit un fichier C$^-$ et qui retourne l'arbre de syntaxe abstraite associé (qui retourne donc une valeur de type `file`).
3. Écrire une fonction `check_scope` qui prend en argument un fichier C$^-$ et qui vérifie que les variables et les fonctions sont bien déclarées quand elles sont utilisées.
4. Écrire une fonction `check_types` qui prend en argument un fichier C$^-$ et qui vérifie que les expressions sont bien typées.

   Pour simplifier, après analyse syntaxique, on ne considérera plus que trois types de base : un pour les entiers, un pour les nombres à virgule flottante, et un pour `void`. De plus, on considérera qu'il n'y a pas de conversion implicite entre ces types ; et que deux types pointeurs seront égaux s'ils pointent vers le même type. Les opérations arithmétiques seront donc bien typées si et seulement si les deux côtés ont le même type soit flottant soit entier (uniquement entier pour `%`). Les opérations de comparaison devront avoir le même type de chaque côté, et retourneront un type entier. Les connecteurs logiques `&&` et `||` devront avoir un type entier de chaque côté et retourneront un type entier. Les opérations d'affectation devront avoir le même type de chaque côté, et retourneront ce type. L'accès à un tableau devra avoir un type t `*` à gauche et un type entier à droite, et retournera le type t. (NB : on ne considérera donc pas l'arithmétique des pointeurs.)