Projet : Analyseurs syntaxique et sémantique pour un sous-langage de TypeScript

Assembleur – Compilation, ENSIIE

Semestre 3, 2024–25

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_tpscrpt.tar.gz . Vous attacherez un soin particulier à ce que ces exemples fonctionnent. (Il y a à la fois des exemples d'entrées correctes et d'entrées que le compilateur est censé rejeter.)

Le projet est à réaliser par groupe de 2.

Votre projet est à déposer sur http://exam.ensiie.fr dans le dépôt `asscom_proj_2024` sur forme d'une archive `tar.gz` **avant le 6 janvier 2025** inclus. **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.

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.

2 Sujet

TypeScript est une extension de JavaScript qui permet de rajouter des annotations de type. Le compilateur TypeScript infère et vérifie les types, puis il produit un code en pur JavaScript (c'est-à-dire sans ces annotations de typage).

Le but de ce projet est d'écrire un exécutable qui prend en entrée un programme écrit en TpScrpt, un sous-langage de TypeScript, qui construit son arbre de syntaxe abstraite

1. Documentation disponible à la page http://dinosaur.compilertools.net/. On peut aussi utiliser les versions libres flex/bison.
2. " http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html

et qui fait quelques analyses sémantiques dessus, puis qui sort du code en pur JavaScript. La syntaxe de TpScrpt est décrite par ce qui suit :

Un commentaire commence soit par `//` et se termine à la fin de la ligne, soit par `/*` et se termine par `*/`. Ce deuxième type de commentaires ne peut pas être imbriqué (donc comme en C et pas comme en OCaml).

Les identifiants en TpScrpt commencent par une lettre (minuscule ou majuscule) ou un underscore, suivi d'un nombre potentiellement nul de lettres, de chiffres et d'underscore, à l'exception des mots-clefs utilisés ci-dessous.

Les constantes numériques seront :

— des constantes décimales entières : une suite non vide de chiffres entre `0` et `9` sans `0` non significatif ; deux chiffres consécutifs pourront éventuellement être séparés par un underscore ;

— des constantes binaires : `0b` ou `0B` suivi d'une suite non vide de chiffres `0` ou `1` ; deux chiffres consécutifs pourront éventuellement être séparés par un underscore ;

— des constantes octales : `0o` ou `0O` suivi d'une suite non vide de chiffres entre `0` et `7` ; deux chiffres consécutifs pourront éventuellement être séparés par un underscore ;

— des constantes hexadécimales : `0x` ou `0X` suivi d'une suite non vide de chiffres entre `0` et `9`, de lettres entre `a` et `f` (minuscule ou majuscule) ; deux chiffres ou lettres consécutifs pourront éventuellement être séparés par un underscore ;

— des constantes décimales flottantes :

  — une constante décimale entière, suivi d'un point, suivi d'une suite potentiellement vide de chiffres entre `0` et `9`, et suivi potentiellement d'un exposant composé d'une lettre `e` ou `E`, éventuellement d'un signe `+` ou `-`, et d'une suite non vide de chiffres entre `0` et `9` ;

  — un point suivi d'une suite potentiellement vide de chiffres entre `0` et `9`, et suivi potentiellement d'un exposant (défini comme dans le cas précédent) ;

  — une constante décimale entière suivi d'un exposant (idem) ;

  dans tous ces cas, deux chiffres consécutifs pourront être séparés par un underscore.

Une constante booléenne est `true` ou `false`.

Une constante chaîne de caractères est un guillemet double `"` suivi d'une suite de caractères différents de `"` sauf s'il est précédé de `\`, suivi de `"` ; ou alors un guillemet simple `'` suivi d'une suite de caractères différents de `'` sauf s'il est précédé de `\`, suivi de `'`.

Un membre gauche est soit :

— un identifiant i

— une expression suivie d'une autre entre crochets e`[`e`]`

— une expression point un identifiant e`.`i

Une expression peut être, par ordre de priorité décroissante des opérateurs (qui se parenthèsent à gauche le cas échéant) :

— une expression entre parenthèses `(`e`)` ;

— une constante (numérique, booléenne ou chaîne de caractères) ;

— un membre gauche ;

— un objet : entre accolades, une suite potentiellement vide de champs, chaque champ étant composé d'un identifiant, d'un deux-points puis d'une expression ; les champs sont séparés par des virgules, le dernier champ peut ou non être suivi d'une virgule `{` i_1`:` e_1`,` `...,` i_n`:` e_n `}` ;

— un tableau constitué d'une suite d'expressions séparées par des virgules, le tout entre crochets ; la dernière expression peut ou non être suivie d'une virgule `[` e_1`,` `...,` e_n `]` ;

— un appel de fonction constitué d'une expression, d'une parenthèse ouvrante, d'une suite potentiellement vide d'arguments composée d'expressions séparées par des virgules ; la dernière expression peut ou non être suivie d'une virgule e`(`e_1`,` `...,` e_n`)` ;

— des opérations unaires : `typeof`, `+` ou `-` suivis d'une expression ;

— une opération d'exponentiation e_1 `**` e_2 ;

— une opération de multiplication e_1 `*` e_2 ou e_1 `/` e_2 ;

— une opération d'addition e_1 `+` e_2 ou e_1 `-` e_2 ;

— une opération de comparaison e_1 `<` e_2 ou e_1 `<=` e_2 ou e_1 `>` e_2 ou e_1 `>=` e_2 ;

— une opération d'égalité e_1 `==` e_2 ou e_1 `!=` e_2 ou e_1 `===` e_2 ou e_1 `!==` e_2 ;

— une opération de conjonction e_1 `&&` e_2 ;

— une opération de disjonction e_1 `||` e_2 ;

— une affectation composé d'un membre gauche, du signe égal `=` et d'une expression l `=` e.

Un type peut-être soit :

— un identifiant ;

— une constante ; par exemple `42` est un type dont le seul élément est `42` ;

— un type primitif `number`, `boolean` ou `string` ;

— le type des tableaux dont les éléments ont le type t qu'on note t`[]` ;

— le type `any` : n'importe quelle valeur peut être de ce type ;

— un type objet : entre accolades, une suite potentiellement vide de déclarations de champs ; chaque déclaration de champ étant composé d'un identifiant, d'un deux-points puis d'un type, ou alors juste d'un identifiant ; les déclaration de champ sont séparées indifféremment par des virgules ou des points-virgules ;

— un type union : une suite de types séparés par `|` ; toute valeur qui possède un de ces types est aussi typable par l'union.

Une instruction est soit :

— l'instruction vide (point-virgule) `;` ;

— une expression suivi d'uns point-virgule e`;` ;

— un bloc constitué d'une suite d'instructions et/ou de déclarations, le tout entre accolades ;

— une déclaration de variable de la forme `var` *bs* `;` où *bs* est une suite non-vide de bindings séparée par des virgules, chaque binding étant de la forme *i to eo* avec *i* un identifiant, *to* est soit vide, soit un deux-points suivi d'un type et *eo* est soit vide, soit `=` suivi d'une expression ;

— une conditionnelle de la forme `if (`*e*`)` i_1 `else` i_2 où *e* est une expression et i_1 et i_2 deux instructions ; la partie else i_2 est optionnelle ; si elle est présente, elle se rapporte au `if` le plus proche ;

— une boucle de la forme `while (`*e*`)` *i* où *e* est une expression et *i* une instruction ;

— un retour `return;` ou alors `return` *e*`;` où *e* est une expression.

Une déclaration peut être :

— un alias de type de la forme `type` *i* `=` *t* où *i* est un identifiant et *t* un type ;

— une déclaration locale au bloc `let` *bs*`;` ou `const` *bs*`;` où *bs* est une suite non-vide de bindings comme ci-dessus ;

— une déclaration de fonction de la forme `function` *i* `(` *as* `)` *to* `{` *dis* `}` où :

  — *i* est un identifiant ;

  — *as* est une suite potentiellement vide de binding comme ci-dessus ;

  — *to* est soit vide, soit `:` suivi d'un type (représentant le type de retour) ;

  — *dis* est une suite potentiellement vide de déclarations et/ou d'instructions.

Un fichier TpScrpt contient une suite potentiellement vide de déclarations et/ou d'instructions.

3 Questions

1. Définir des types de données correspondant à la syntaxe abstraite des programmes TpScrpt (couvrant toute la grammaire). En particulier on définira entre autres des types `type_`, `expression`, `instruction` et `declaration`.
2. À l'aide de lex/yacc, ou ocamllex/ocamlyacc, écrire un analyseur lexical et syntaxique qui lit un fichier TpScrpt et qui retourne l'arbre de syntaxe abstraite associé.
3. Écrire une fonction `check_scope` qui prend en argument un AST de fichier TpScrpt et qui vérifie que les identifiants ont bien été déclarés quand ils sont utilisés (variables et appels de fonctions).

   La portée des variables déclarées par `var` est celle de la fonction où elles sont déclarées. En particulier, on peut les utiliser et les affecter avant la déclaration.

   La portée des variables déclarées par `let` ou `const` est celle du bloc où elles sont déclarées. De plus, on ne peut les utiliser et les affecter qu'après la déclaration.

   On vérifiera également qu'on ne fait pas d'affectation dans les variables déclarées par `const`.

4. Écrire une fonction `check_type` qui prend en argument un AST de fichier TpScrpt et qui vérifie que les expressions sont bien typées.

   Dans une instruction e`;`, e peut avoir n'importe quel type, qui est ignoré.

   Si le type de retour d'une fonction n'est pas précisé, c'est `any`. L'expression e d'un `return` e`;` doit être du type de retour de la fonction. Un `return` sans expression n'est possible que si le type de retour est `any`.

   Si on accède à un champ `x.f`, si le type de `x` est une union, il faut vérifier que chacun des membres de cette union possède ce champ.

   Dans les déclarations des variables, si le type n'est pas donné explicitement, il est inféré à partir de la valeur donnée pour initialiser. Dans un premier temps, on pourra considérer que c'est `any`.

   On a une relation de sous-typage : le type `42` est un sous-type de `number` qui est un sous-type de `any`. De même, `"coucou"` est un sous-type de `"coucou" | number | { x : string; y }`.

5. Écrire une fonction qui prend en argument un AST et qui l'affiche sur la sortie standard, mais sans les informations de typage, pour avoir du code JavaScript pur.

6. Écrire un programme qui parse un fichier TpScrpt, effectue les vérifications `check_scope`, et `check_type` en renvoyant un code d'erreur différent de 0 au cas où ils ne passent pas, puis qui affiche la version JavaScript pur sur la sortie standard. Afficher des messages expliquant les erreurs sera fortement apprécié.