Projet : Analyseur syntaxique et sémantique pour Pseudo-Pascal

Assembleur – Compilation, ENSIIE

Semestre 3, 2022–23

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://www.ensiie.fr/~guillaume.burel/compilation/projet_pp_synsem.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 `asco_projet` sur forme d'une archive `tar.gz` **avant le 4 janvier 2023 à 23h59**. **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.

2 Sujet

Le but de ce projet est d'écrire un programme qui prend en entrée un programme Pseudo-Pascal et qui construit son arbre de syntaxe abstraite, puis qui fait une analyse sémantique dessus pour vérifier les types et la portée. La syntaxe de Pseudo-Pascal est décrite par ce qui suit :

Un programme Pseudo-Pascal est de la forme

```
program
  définition de variables globales (optionnel)
  définitions de fonctions/procédures (optionnel)
begin
  instructions séparées par ;
end.
```

Les définitions de fonctions sont de la forme

```
function identifiant (environnement) : type;
  définition de variables locales (optionnel)
begin
  instructions séparées par ;
end;
```

1. Documentation disponible à la page `http://dinosaur.compilertools.net/`
2. " `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html`

Les définitions de procédures sont de la forme

```
procedure identifiant (environnement);
  définition de variables locales (optionnel)
begin
  instructions séparées par ;
end;
```

Un environnement est une suite d'éléments séparés par des `;`, où chaque élément est de la forme *liste d'identifiants séparés par des* `,` `:` *type*.

Un identifiant est composé d'un caractère alphabétique suivi d'un nombre quelconque de caractères alphanumériques. Un type est soit `integer` soit `boolean` soit (`array of` suivi d'un type).

Les définitions de variables (globales ou locales) sont de la forme `var` *environnement non vide* `;`.

Les instructions peuvent être

- appel de procédure : *identifiant*`(`*liste d'expressions séparées par* `,` `)`
- affectation dans une variable : *identifiant* `:=` *expression*
- affectation dans un tableau : *expression*`[`*expression*`]` `:=` *expression*
- conditionnelle : `if` *condition* `then` *instruction* `else` *instruction*
- boucle non bornée : `while` *condition* `do` *instruction*
- bloc : `begin` *liste d'instructions séparées par* `;` `end`

En plus de ceci, on autorisera une autre forme de boucle :
`repeat` *listes d'instructions séparées par* `;` `until` *condition*
Néanmoins, on n'aura pas de nœuds spécifique pour cela dans l'arbre de syntaxe abstraite, car `repeat` l `until` c sera du sucre syntaxique pour

```
begin
  l;
  while c do
    begin
      l
    end
end
```

Les conditions peuvent être *expression* ou `not` *condition* ou *condition* `or` *condition* ou *condition* `and` *condition* ou `(`*condition*`)`. `and` est prioritaire sur `or`.

Les expression peuvent être

- constante : *entier* ou `true` ou `false`
- variable : *identifiant*
- moins unaire : `-` *expression*
- opérations arithmétiques : *expression op expression* où *op* est l'un de `+ - * /` (avec priorités usuelles)
- comparaison : *expression comp expression* où *comp* est l'un de `< <= > >= = <>`
- appel de fonction : *identifiant*`(`*liste d'expressions séparées par* `,` `)`
- accès dans un tableau : *expression*`[`*expression*`]`
- création d'un tableau : `new array of` *type*`[`*expression*`]`

Entre tous lexèmes d'un programme Pseudo-Pascal peuvent apparaître des commentaires entre accolades `{` et `}`.

3 Questions

1. Définir des types de données correspondant à la syntaxe abstraite des programmes Pseudo-Pascal (couvrant toute la grammaire). En particulier on définira entre autres des types `program`, `instruction`, `condition` et `expression`.
2. À l'aide de lex/yacc, ou ocamllex/ocamlyacc, écrire un analyseur lexical et syntaxique qui lit un programme Pseudo-Pascal et qui retourne l'arbre de syntaxe abstraite associé (qui retourne donc une valeur de type `program`).
3. Écrire une fonction `check_scope` qui prend en argument un programme Pseudo-Pascal et qui vérifie que les variables et les fonctions sont bien déclarées quand elles sont utilisées. On rappelle que la valeur de retour d'une fonction est donnée à l'aide d'une variable spéciale utilisant le même identifiant que la fonction.

   On considérera que les procédures `write` et `writeln` et la fonction `readln` sont prédéfinies ; elles ne doivent donc pas générer d'erreur de portée.
4. Écrire une fonction `check_types` qui prend en argument un programme Pseudo-Pascal et qui vérifie que les expressions, conditions et instructions sont bien typées. (NB : les indices et les tailles des tableaux sont forcément des `integer`.)

   On ne vérifiera pas les types des paramètres de `write` et `writeln` ; la function `readln` ne prendra pas d'arguments.
5. Écrire un programme qui parse un fichier Pseudo-Pascal, puis effectue les vérifications `check_scope` et `check_types`, en renvoyant un code d'erreur différent de 0 au cas où ils ne passent pas.
6. Bonus : écrire des messages d'erreurs informatifs (position de l'erreur dans le fichier, raison de l'erreur) en cas d'erreurs de syntaxe, de portée ou de typage.