Projet de Compilation: Graphe d'appel Pseudo-Pascal

ÉNSIIE, semestre 3, 2013–14

1 Informations pratiques

Ce projet est à effectuer en binômes. Dès qu'un binôme se sera constitué, il enverra un mail à `guillaume.burel@ensiie.fr` pour vérification.

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 depuis l'adresse : `http://www.ensiie.fr/~guillaume.burel/compilation/` . Vous attacherez un soin particulier à ce que ces exemples fonctionnent.

Votre projet est à envoyer sur forme d'une archive `tar.gz` **avant le 20 décembre 2013 à 18h** par mail à `guillaume.burel@ensiie.fr`. **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 graphe d'appel d'un programme est un graphe orienté dont les nœuds sont les procédures et fonctions du programme et tel qu'il y a une arête de `f` vers `g` s'il y a un appel à `g` dans le corps de `f`. Le but de ce projet est d'écrire un programme qui prend en entrée un programme Pseudo-Pascal et qui affiche sur la sortie standard son graphe d'appel en suivant la syntaxe de `dot` (cf. section 2.3). On distinguera les phases suivantes :

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

2.1 Front-end : analyse syntaxique

L'objectif de cette phase est de reconnaître un fichier contenant un programme Pseudo-Pascal et de produire l'arbre de syntaxe abstraite correspondant.

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 [3] sont de la forme

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

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 : `while` *condition* `do` *instruction*
- bloc : `begin` *liste d'instructions séparées par* `;` `end`

3. Bien que ce ne soit pas utile pour ce projet, la valeur de retour de la fonction est donnée à l'aide d'une variable spéciale du même identifiant que la fonction.

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 tout lexème d'un programme Pseudo-Pascal peuvent apparaître des commentaires entre accolades `{` et `}`.

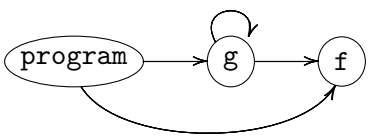
1. Définir des types de données correspondant à la syntaxe abstraite des programmes Pseudo-Pascal (couvrant toute la grammaire). En particulier on définira entre autres des types `program`, `instruction` et `expression`.
2. À l'aide de lex/yacc, ou ocamllex/ocamlyacc, écrire un analyseur lexical et syntaxique qui lit un programme Pseudo-Pascal et qui retourne l'arbre de syntaxe abstraite associé (qui retourne donc une valeur de type `program`).

2.2 Middle-end : calcul du graphe d'appel

Le graphe d'appel statique d'un programme est un graphe orienté dont les nœuds sont les fonctions et procédures du programme et dont les arêtes indiquent qu'une fonction/procédure en appelle potentiellement une autre. Un nœud spécial dont l'étiquette sera `program` représentera la procédure principale du programme. Par exemple le programme suivant :

```
program
function f () : integer;
begin
  f := 1
end;
procedure g(x : integer);
begin
  if x >= 0 then x := f() else g(x + 1)
end;
begin
  g(f())
end.
```

a pour graphe d'appel



On remarquera que ce graphe est une surapproximation du graphe d'appel réel (dynamique) du programme : pour le programme ci-dessus on a mis une flèche de `g` vers `g` correspondant à la branche `else` bien que cette branche ne soit pas exécutée par le programme. Cela provient du fait que le calcul du graphe d'appel réel est en général indécidable.

3. Définir un type de donnée `call_graph` pour les graphes d'appel. On pourra par exemple utiliser une map qui associe à des identifiants des ensembles d'identifiants.
4. Écrire une fonction `make_call_graph` qui prend en entrée l'arbre de syntaxe abstraite d'un programme Pseudo-Pascal et qui retourne son graphe d'appel. On pourra définir des fonctions auxiliaires `make_call_graph_instr` et `make_call_graph_expr` qui analyse uniquement une instruction (resp. une expression).

2.3 Back-end : sortie `dot`

`dot` est un langage de description de graphe sous forme de texte. Les fichiers au format `dot` peuvent ensuite être transformés par les outils de la suite Graphviz (notamment le programme `dot`) pour fournir des représentations graphiques du graphe, au format SVG, PDF, PNG, etc. Un fichier représentant un graphe orienté au format `dot` ressemble à

`digraph` *nom* `{`
  liste de déclarations terminées par `;`
`}`

où une déclaration est soit la déclaration d'un nœud, en donnant uniquement son étiquette suivi de `;` (pas obligatoire si le nœud apparaît dans une arête), soit la déclaration d'une arête `noeud1 --> noeud2;`. Il est possible d'ajouter des attributs aux nœuds et aux arêtes en les mettant entre crochet avant le `;`, par exemple `[color=red]`. On se référera au manuel de Graphviz [4] pour connaître la liste des attributs disponibles.

5. Écrire une fonction `print_call_graph` qui prend en entrée un graphe d'appel et qui l'affiche au format `dot` sur la sortie standard.
6. Écrire une fonction principale qui lit un fichier au format Pseudo-Pascal sur l'entrée standard et qui affiche son graphe d'appel au format `dot` sur la sortie standard.
7. Tester votre programme sur les fichiers d'entrée disponibles depuis la page `http://www.ensiie.fr/~guillaume.burel/compilation/`.

Bonus : Un appel à `g` dans la fonction `f` est dit terminal si c'est la dernière opération effectuée dans `f`. La compilation de ces appels peut être optimisée (cf. cours), d'où l'intérêt de les identifier.

8. Modifier la structure de données des graphes d'appel pour pouvoir y indiquer qu'un appel est terminal ou non.
9. Modifier les fonctions `make_call_graph` et `print_call_graph` en conséquence. On pourra par exemple utiliser des arêtes en pointillés (`[style=dashed]`) pour représenter les appels terminaux.

4. `www.graphviz.org/Documentation/dotguide.pdf`