Compilation de C−−: phase d'analyse

Projet « Programmation 1 »

Sur la deuxième partie du premier semestre et le debut du second semestre, vous aurez comme objectif de coder un compilateur pour un sous-ensemble du langage C, que nous appellerons le langage C−−. Ce compilateur devrait prendre en entrée un fichier contenant du code en C−−, contrôler que le code est correct (c'est-à-dire, qu'il respecte les types et les règles statiques du langage), et générer du code LC3 [1].

La phase d'analyse doit être codée pendant la deuxième partie du premier semestre. La date limite du *rendu complet* de cette partie est *le vendredi 20 décembre 2024*. Cette phase se décompose en plusieurs étapes :

I. L'analyse lexicale et l'analyse syntaxique sont décrites dans ce document et elles sont déjà codées dans les sources qui vous sont distribuées.

II. Le typage est décrit dans ce document. Le rendu intermédiaire pour cette étape doit être fait *au plus tard le vendredi 22 novembre 2024*.

III. La création du graphe de flot de contrôle (Control Flow Graph – CFG) et trois analyses utilisant le CFG (décrites dans un prochain document) :

   a. l'analyse de constantes,

   b. l'analyse de code « mort » (non accessible),

   c. l'analyse d'initialisation (une variable est déjà initialisée avant son utilisation).

   Le rendu intermédiaire pour cette étape doit être fait *au plus tard le vendredi 13 décembre 2024*.

Les rendus intermédiaires seront utilisés par les encadrants du projet afin de vous faire des retours sur votre projet et d'apprécier l'avancement de votre travail. Le rendu final et ceux intermédiaires seront faits sur la forme d'une archive compressée (`.zip` ou `.tar.gz`) d'un répertoire contenant les sous-repertoires suivants :

— `src` : vous y déposerez le code OCaml de votre projet ainsi que les fichiers utilisées pour le compiler,

— `tst` : les programmes écrits en C−− (extension de fichier `.c`) que vous avez utilisés pour tester votre compilateur. Chaque programme doit être accompagné d'un commentaire ou d'un fichier ayant le même nom mais l'extension `.log` contenant les résultats de la phase d'analyse.

— `doc` : la documentation (en format textuel Markdown ou LaTeX) qui décrit l'architecture de votre code, les choix de codage, les résultats obtenus, les tests utilisés.

L'archive distribuée sur eCampus avec le code de la première étape respecte cette architecture. Il faut la préserver et s'inspirer pour les étapes que vous coderez.

La phase de synthèse (génération de code LC-3) sera le sujet des premières séances du projet du module « Langages formels » au second semestre.

Ce document explique les restrictions de lexique et syntaxe du C−− ainsi que les le règles de typage à coder pour la première étape d'analyse.

1. `http://lc3tutor.org/`

1 Syntaxe

Le C−− contient des notions déjà vues dans le langage C (variable, fonction, constantes entières, type entier et adresse) mais beaucoup plus simplifiés. Les principales restrictions sont présentées dans cette section.

1.1 Lexique et analyse lexicale

Le langage C−− ne contient qu'un ensemble restreint de constantes, les constantes entières. Les constantes entières peuvent être écrites en base 10, 8 ou 16. Il résulte que les seuls types utilisables sont le type `int` et les types adresse. Le type adresse peut faire référence à un type `int`, à un type fonction, ou à un autre type adresse. Les déclarations multiples de variables ne sont pas permises ; les variables peuvent être initialisées à la déclaration.

Les opérateurs arithmétiques et de comparaison sont ceux du C. Les opérateurs logiques sont admis, mais pas les opérateurs bit-à-bit. Les expressions peuvent contenir des appels de fonction. L'expression conditionnelle (`(a)?b:c`) est aussi autorisée.

Les commentaires et les identificateurs sont ceux du C.

Les instructions sont restreintes à l'affectation, décision (`if`, `if ... else`) et aux boucles (`for` et `while`). Les instructions de saut (`break`, `continue`, `goto`) ne sont pas présentes.

Le lexique du C−− est formalisé dans le fichier `src/clexer.mll`. Il est compilé avec `ocamllex` pour générer le code OCaml de l'analyseur. Cette façon de compiler est spécifiée dans le fichier `dune` par : `(ocamllex (modules clexer))`.

L'analyseur lexical est appelé dans le fichier `main.ml`, qui contient les appels aux phases principales de la compilation.

1.2 Syntaxe et analyse syntaxique

La syntaxe de C−− est réduite aux constructions du C qui utilisent le lexique restreint présenté dans la section précédente. La seule différence est pour la déclaration d'une variable de type adresse de fonction, dont un exemple (correct) est donné dans le fichier `tst/test03.c`.

L'analyseur syntaxique du C−− est codé dans le fichier `src/cparser.mly`, compilé avec `menhir` pour générer le code OCaml de l'analyseur. Cette façon de compiler est spécifiée dans le fichier `dune` par : `(menhir (modules cparser))`.

L'analyseur syntaxique construit un arbre de syntaxe abstraite (AST), valeur de type `file` dans le fichier `src/cast.mli`.

L'analyseur syntaxique est appelé dans le fichier `main.ml`.

Le code distribué permet à présent de transformer des fichiers source écrits en C−− vers des fichier en format DOT contenant l'arbre de syntaxe abstraite. Pour cela, il faut :

1. nettoyer les fichiers générés avec : `dune clean`
2. compiler les sources avec : `dune build`
3. appeler l'exécutable généré avec : `./ccomp --debug test.c`

Ces commandes sont en partie présentes dans le fichier `Makefile` du sous-répertoire `src`.

Affichage de l'arbre syntaxique. Afin de vous aider à lire les AST, nous fournissons un module `src/pretty.ml` qui offre une fonction pour traduire un AST vers le format DOT dans un fichier `.dot`. L'AST en format DOT peut être ensuite visualisée avec l'outil `dot` du paquetage `graphviz` par `dot -v -Tpng -O test_ast.dot`. Un raccourci pour ceci est défini dans le `Makefile`.

Attention, vous devez modifier uniquement les fichiers qu'on vous demande à modifier dans le fichier `README.md`.

2 Typage statique

Les règles de typage informelles sont :

1. Les types autorisés sont : entier et pointeur vers un type (par exemple, `int*`, `int**`) et les pointeurs de fonction (par exemple `int(*)(int,int)`).
2. Une variable globale est déclarée une seule fois.
3. Une fonction est définie une seule fois.
4. La définition d'une fonction ne doit pas avoir deux paramètres avec le même nom.
5. Une variable locale peut être redéfinie dans un bloc imbriqué.
6. Une variable locale peut avoir le nom d'une fonction, mais alors elle cache la définition de la fonction.
7. Les valeurs affectées doivent être de type de la valeur gauche de l'affectation.
8. L'opérateur « adresse de » (`&`) ne peut être appliqué que sur une variable qui a un type entier ou adresse d'un tel type.
9. La conversion implicite entre type adresse et type entier n'est pas permise.
10. Entre les valeurs entières, les opérations arithmétiques présentes dans la syntaxe sont autorisées.
11. Entre les valeurs de type adresse du même type qui n'est pas un type fonction, la seule opération permise est la soustraction [2].
12. Entre une valeur de type adresse d'un type qui n'est pas fonction et une valeur entière, les opérations permises sont : l'addition et la soustraction.
13. Une fonction doit être appelée avec le type et le nombre des paramètres correspondant à sa définition.
14. Le bloc d'une fonction doit contenir une instruction `return` avec une expression du type de retour déclarée pour la fonction, dans toutes les branchements du code.
15. Deux fonctions `print_int` et `print_string` sont disponibles pour afficher respectivement un entier et un texte.

Ces règles doivent être codées dans le fichier `ctyping.ml`, dans une fonction `check_file`. Cette fonction renvoie la liste des variables globales et fonctions déclarées, chaque avec leur type si le code analysé est bien typé et lève une exception sinon ; elle signale les erreurs de typage par des messages faciles à interpréter.

2. Ceci est une simplification par rapport à la norme C qui demande que les pointeurs soient dans le même bloc mémoire.