Projet de compilation en licence informatique (UNS, 2018): mini compilateur C

Pr Sid Touati, Mme Cinzia Di Giusto, Mme Elisabetta Di Maria

2018

Table des matières

Le projet est à réaliser par binômes. Il doit être déposé dans l'espace de cours sur `jalon.unice.fr` dans la boîte de dépôt prévue à cet effet. Pas de soumission par mel. Vous devrez faire une courte soutenance de 10 minutes, pas de rapport demandé.

Il est demandé de réaliser un compilateur, à l'aide des outils **lex** et **yacc**. Votre archive de projet devra être nommée **Projet-Nom1-Nom2.tar.gz**. Elle doit contenir tous les sources de votre compilateur, les descriptions lex et yacc, ainsi que les programmes de tests. Il est demandé de fournir un fichier `LisezMoi.txt` précisant comment compiler et tester votre compilateur, et les problèmes éventuels (pour que le correcteur sache à l'avance si votre projet fonctionne ou pas). Un jeu de tests (programmes C que votre compilateur devra compiler) vous est fourni dans le répertoire adéquat.

Votre archive de projet **Projet-Nom1-Nom2.tar.gz** doit être soumise via la boite de dépôt de jalon, aucune soumission par mail ne sera acceptée.

1 Présentation générale du projet

Le but de ce projet est de réaliser un mini compilateur C (pour un sous langage de C).Votre compilateur prendra en entrée un code C simplifié et génèrera un code trois adresses dans un langage destination (défini ultérieurement). Le langage C simplifié en entrée s'appelle Cfe (C-frontend) et le langage en sortie Cbe (C-backend).

Le langage Cbe est également un sous-ensemble de C, mais plus proche de l'assembleur. Ainsi, un code écrit en Cfe ou en Cbe peut être compilé et testé avec n'importe quel compilateur C comme gcc.

2 Langage source : Cfe

On considère le langage Cfe dont les caractéristiques sont les suivantes. Il s'agit d'un sous-ensemble de C qui doit pouvoir être compilé avec un compilateur C classique comme gcc.

2.1 Description du langage Cfe

- Structure du programme :
  - la structure d'un programme est la suivante :
    $<declarations>$
    $<fonctions>$
  - toute variable doit être déclarée avant utilisation.
- Les identificateurs :
  - ils ne peuvent pas porter le nom d'un mot clé réserve : **extern, int, void, for, while, if, then, else, switch, case, default, break**.
- Les variables :

  - elles ne peuvent être déclarés qu'en début de programme (variables globales) ou qu'au début d'un bloc (variables locales).
  - elles doivent commencer par une lettre,
  - elles sont de type entier ou tableau d'entiers à un nombre quelconque de dimensions.
- Les constantes :
  - elles sont entières et exprimées en base 10.
- Les fonctions :
  - structure d'une fonction :

<type> identificateur (<liste_parametres>)
{
<bloc>
}

  - **identificateur** est le nom de la fonction
  - le **type** d'une fonction est soit **int** soit **void**
  - un seul niveau de déclaration de fonctions est possible (pas d'imbrications de déclarations)
  - les paramètres sont des entiers
  - une fonction peut être déclarée **extern**. Dans ce cas, elle est déclarée mais pas définie/implémentée dans le programme (son code est dans un autre fichier). Cela permet de faire référencer des fonctions de librairie ou écrites dans un autre fichier tout en permettant de vérifier que toutes les fonctions utilisées soient déclarées.
- Les instructions considérées sont :
  - L'affectation *var = expression*
  - les structures de contrôle
  - **return**. Elle peut retourner une valeur ou non.
- Les instructions de contrôle ont la même sémantique que celle du C :
  - l'instruction **if..then** et **if..then..else**
  - l'instruction **switch**, avec **case** et **default**,
  - l'instruction **for**
  - l'instruction **while**
  - l'instruction **break**
- Les conditions :
  - une condition est toujours entre parenthèses
  - les opérateurs de comparaison dont on dispose sont : >, <, <=, >=, ==, **!=**
  - les opérateurs booléens sont : **!**, **&&**, **||**
- Les expressions :
  - les opérateurs binaires dont on dispose sont : +, **-**, *****, /, «, », **&**, **|**
  - une expression est composée de valeurs numériques et/ou de variables et d'opérateurs binaires.
- Les commentaires :
  - les commentaires débutent par /* et se terminent par */
  - ils peuvent être sur plusieurs lignes

2.2 Exemple d'un programme écrit en Cfe

Le code `exempleCfe.c` est fourni en exemple. Vous pouvez tester que gcc le compile correctement avec la commande `gcc -c exempleCfe.c`, cela produira un code objet.

2.3 Grammaire du langage Cfe

La grammaire du langage Cfe est décrite ici. Nous fournissons une description yacc de départ dans le fichier appelé `cfe.y`. Vous pouvez modifier ou compléter cette description yacc pour le besoin de votre projet.

Également, nous fournissons une description lex du langage C dans le fichier nommé `ANSI-C.l`. Vous pouvez modifier ou compléter cette description lex pour la rendre correcte vis à vis du langage Cbe ; par exemple, les mots clés du langage C qui n'appartiennent pas au langage Cbe doivent être enlevés, etc.

3 Langage destination : Cbe

3.1 Description du langage Cbe

Le langage destination Cbe est un sous-ensemble du C, proche de l'assembleur. La structure du programme est la même que Cfe et les déclarations de variables et fonctions se font de la même façon. Les types permis pour les variables sont **int** et **int ***. De plus, les instructions doivent correspondre à du code 3 adresses. Un code 3 adresses est une séquence d'instructions qui sont de la forme : **x = y op z**, où **x, y et z** sont des noms, des constantes ou des variables temporaires introduites par le compilateur ; **op** désigne n'importe quel opérateur. Un programme Cbe doit pouvoir être compilé avec n'importe quel compilateur C comme gcc.

Une description plus complète est donnée ci-dessous.

- Structure du programme :
  - la structure d'un programme est la suivante :

    <declarations>
    <fonctions>
  - toute variable doit être déclarée avant utilisation.
- Les identificateurs :
  - ils ne peuvent pas porter le nom d'un mot clé réserve : **extern, int, void, if, goto**.
- Les variables :
  - elles ne peuvent être déclarées qu'en début de programme (variables globales) ou qu'au début d'un bloc (variables locales).
  - elles sont de type entier ou pointeur sur entier.
- Les constantes :
  - elles sont entières et exprimées en base 10.
- Les fonctions :

- structure d'une fonction :

  <type> identificateur (<liste_parametres>)
  {
  <bloc>
  }
- **identificateur** est le nom de la fonction
- le **type** d'une fonction est soit **int** soit **void**
- un seul niveau de déclaration de fonctions est possible (pas d'imbrications de déclarations)
- les paramètres sont des entiers
- une fonction peut être déclarée **extern**. Dans ce cas, elle est déclarée mais pas définie/implémentée dans le programme (son code est dans un autre fichier). Cela permet de faire référencer des fonctions de librairie ou écrites dans un autre fichier tout en permettant de vérifier que toutes les fonctions utilisées soient déclarées.

— Les instructions considérées sont :
- les appels de fonctions, où les paramètres ne sont que des noms de variables ou des constantes.
- les seules instructions d'affectation autorisées sont de la forme suivante, avec **x** un nom de variable, **y** et **z** des constantes ou des noms de variables et **t** un nom de tableau ou de pointeur sur entier :
  - **x = y op z**
  - **x = - y**
  - ***t = z**
  - **x = *t**
  - **x = f(...)** avec f une fonction dont les arguments ne sont que des constantes ou des noms de variables.
- le branchement inconditionnel : de la forme **goto L** avec L un nom de label déclaré par **L :**
- le branchement conditionnel : de la forme **if (x op y) goto L** ou **if (!x) goto L** avec **x, y** des noms de variables ou des constantes, **op** un opérateur binaire
- Les noms des labels (ou étiquettes) doivent respecter les mêmes règles que les identificateurs.
- l'instruction **return**. Elle peut retourner une valeur sous la forme d'un nom de variable ou d'une constante.

— Les commentaires :
- les commentaires débutent par /* et se terminent par */
- ils peuvent être sur plusieurs lignes

3.2 Exemple d'un programme écrit en Cbe

Le code `exempleCbe.c` est fourni en exemple. Vous pouvez tester que gcc le compile correctement avec la commande `gcc -c exempleCbe.c`, cela produira un code objet.

3.3 Grammaire du langage Cbe

La grammaire du langage Cbe est décrite ici. Nous fournissons sa description yacc dans le fichier appelé `cbe.y`. Vous pouvez modifier ou compléter cette description yacc pour votre besoin. Si vous modifiez la grammaire, expliquez nous pourquoi l'avez vous fait durant la soutenance, et également dans le fichiez `LisezMoi.txt` que vous fournirez.

Pour une description lex du langage Cbe, vous pouvez démarrer avec le fichier nommé `ANSI-C.l`. Vous pouvez modifier ou compléter cette description lex pour la rendre correcte vis à vis du langage Cbe. Les descriptions lex et yacc du langage Cbe ne sont pas nécessaires pour réaliser votre compilateur, par contre ils vous serviraient pour générer un parseur pour le langage Cbe, afin de vérifier que votre code généré est correct lexicalement et syntaxiquement.

4 Traduction de Cfe vers Cbe

Le but de ce projet est de faire le compilateur du langage d'entée Cfe vers le langage Cbe. La traduction de Cfe vers Cbe oblige à introduire de nouvelles variables afin de respecter les instructions 3 adresses. Le compilateur pourra introduire un nombre illimité de variables. Les nouvelles variables générées devront commencer par un caractère souligné et devront être déclarées en tant que variables locales, avec le type approprié.

D'autre part, comme les tableaux multidimensionnels ne sont pas autorisés en Cbe, les tableaux multidimensionnels de Cfe devront être linéarisés (transformés en tableaux d'une dimension).

Etant donné que l'on veut se rapprocher le plus possible du modèle "assembleur", on impose les traductions suivantes pour les structures de contrôle du langage Cfe :

— La boucle **while** devient :

goto L1
L2 : < corps_de_la_boucle >
L1 : if (< test >) goto L2

— L'instruction **if** devient :

if (! < test >) goto L
< instruction(s) > / du then */*
L : < suite_du_programme >

— L'instruction **switch** peut être traduite en une série de comparaisons (entre la valeur de l'expression et des cases) et de **goto**

— L'instruction **break** peut être traduite en un **goto** vers l'instruction suivant immédiatement le bloc courant. Notez que le **break** peut être utilisé ailleurs que dans un **switch**

— Un exemple de traduction de la boucle **for** est donné dans le code `exempleCbe.c`, qui est la traduction en Cbe du programme précédent en Cfe. Les variables crées par le compilateur ont des noms qui commencent par un souligné. `exempleCbe.c` est une traduction du code `exempleCfe.c` , les deux codes devraient fournir le même résultat. [1].

5 Conseils pratiques

Un projet de compilation est très important dans la vie d'un étudiant en informatique. Pour beaucoup d'entre vous, c'est probablement la seule occasion dans votre carrière d'écrire un compilateur. Il est donc important de faire des efforts pour le réaliser jusqu'au bout, car c'est travail très formateur que des étudiants des autres disciplines scientifiques ne font pas.

Vous êtes autorisés à communiquer entre vous tous (tous les étudiants), à vous échanger des idées et des astuces, à vous entre-aider pour comprendre et progresser ensemble. En revanche il est interdit de soumettre un projet qui n'est pas le fruit de votre travail : interdiction formelle de copier le code d'une autre personne, ou qu'une personne fasse le projet à la place d'une autre personne. Ceci constitue une fraude.

Concernant le travail par binôme, nous ne voulons pas qu'un des deux membres fasse tout le projet pour l'autre. Il faut apprendre à travailler en équipe pour apprendre. Les enseignants sont capables de discerner si un étudiant a fourni des efforts ou pas dans un travail de binôme, et peuvent décider de donner des notes différentes.

Par expérience, les étudiants trouvent un projet de compilation difficile. C'est pas faux. Ci-dessous quelques conseils :

— Le projet de compilation est un effort sur tout le semestre. Commencez le dès que vous abordez l'analyse lexico-syntaxique en TD. N'attendez pas les dernières semaines du semestre pour commencer votre projet, car vous découvrirez des difficultés techniques.

— Commencez par tester votre analyseur lexical en premier lieu, et vérifiez que tous les programmes tests fournis passent correctement (à savoir que les tokens sont correctement analysés par votre analyseur lexical). Vérifiez sa robustesse en modifiant les programmes tests pour provoquer des erreurs lexicales : introduisez des caractères interdits, des noms de variables incorrects, etc.

— Une fois que vous êtes sûrs que votre analyseur lexical fonctionne correctement, vérifiez que votre parseur (analyseur lexico-syntaxique) fonctionne sur tous les programmes tests. Vérifiez sa robustesse en modifiant les programmes tests pour provoquer des erreurs syntaxiques.

1. Le code de la fonction `printd` se trouve dans le fichier `Tests/printd.c`

— Une fois que vous estimez que votre parseur est fiable, commencez à introduire des routines sémantiques yacc qui affichent des messages appropriés durant la compilation afin de tester que les routines sémantiques s'exécutent correctement. Ce sera le squelette de votre traduction dirigée par la syntaxe, que vous devrez compléter pour la génération de code.
— Attention, en langage C, contrairement à d'autres langages, les chaines de caractères doivent être allouées en mémoire explicitement. Si une routines sémantique de yacc modifie une chaine de caractères non allouée en mémoire, votre compilateur plantera.

6 Notation du projet

Le projet sera noté sur 20 points, 5 points pour la soutenance et 15 points pour le compilateur. La note du projet comptera pour 50% de la note finale en compilation, l'examen terminal comptera pour les 50% restant.

Pour la soutenance, il est demandé de présenter votre travail de façon concise avec des slides : les difficultés rencontrées et comment vous les avez traitées, quelles sont les structures de données employées dans votre compilateur, ce que fait au final votre compilateur et ce qu'il ne fait pas, etc. C'est un moment d'échange où il faut être honnête avec l'enseignant. La durée sera de 10 min (5-6 slides), il est recommandé de s'entrainer avant ! La soutenance ne devra pas être un récital des notions étudiées en cours.

Consultez le fichier `Tests/Lisezmoi.txt` pour savoir comment allons nous procéder pour tester votre projet. Concernant la notation de votre compilateur (sur 15 points), ci-dessous quelques précisions :

— Si aucun code source n'est fourni dans l'archive de votre projet, la note sera 0.
— Si le code source de votre compilateur ne compile pas avec gcc, *i.e.* si l'enseignant n'arrive pas à générer l'exécutable de votre compilateur, la note sera en dessous de 7,5.
— Si votre compilateur s'exécute sur des programmes tests, mais il plante, sa note sera en dessous de 7,5.
— Si votre compilateur s'exécute sur tous les programmes tests sans planter, mais ne génère aucun code (ie votre compilateur est juste un parseur), la note sera en dessous de 7,5.
— Si votre compilateur s'exécute sur des programmes tests, mais génère des codes faux pour certains et des codes corrects pour d'autres, la note sera supérieure à 7,5.
— Si votre compilateur s'exécute correctement sur tous les programmes tests et qu'il génère du code correct pour tous les programmes tests, la note sera 15.

Nous vous souhaitons tout le succès.

Pr Sid Touati, Mme Cinzia Di Giusto, Mme Elisabetta Di Maria