Projet de compilation en licence informatique: `strucit`, un mini compilateur C

Pr Sid Touati
Université Côte d'Azur

2024

Table des matières

1 Consignes sur la soumission et notation du projet de compilation

Le fichier `"Grille d'évaluation du projet de compilation"` mis en ligne dans l'espace internet de l'UE de compilation indique avec précision quoi faire pour soumettre votre projet et comment celui-ci sera noté. Suivre les consignes demandées.

2 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. Le langage C simplifié en entrée s'appelle **STRUCIT-frontend** et le langage en sortie **STRUCIT-backend**. Un code écrit dans le langage STRUCIT-frontend peut être compilé et testé avec n'importe quel compilateur C comme gcc. Le langage STRUCIT-backend est un dialecte du C, plus proche de l'assembleur (non accepté par un compilateur C). En résumé, vous devrez réaliser deux programmes principaux :

1. Un compilateur qui traduit un code du langage STRUCIT-frontend vers un code STRUCIT-backend, et affiche des messages d'erreur si le code en entrée est incorrect.
2. Un simple parseur pour le langage STRUCIT-backend, qui permet de vérifier si un code écrit en langage STRUCIT-backend est correct lexicalement et syntaxiquement (pas d'analyse sémantique demandée pour ce parseur). Il devra afficher des messages d'erreur sinon. La vérification sémantique du code généré STRUCIT-backend se fera à la main.

3 Langage source : STRUCIT-frontend

On considère le langage STRUCIT-frontend 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`. Les contraintes syntaxiques et sémantiques sont les mêmes que celles du C, avec quelques contraintes et restrictions en plus qui sont détaillées ici.

3.1 Description du langage STRUCIT-frontend

- Les identificateurs :
  - ils doivent commencer par une lettre, et ne comportent que des lettres, chiffres et soulignés ;
  - ils ne peuvent pas porter le nom d'un mot clé réservé : **extern, int, void, for, while, if, else, return, struct, sizeof**.
- Types apparaissant dans le programme :
  - types de base : **int**, **void** ;

- types construits :
  - structures : les structures sont construites avec le mot clé **struct** ;
  - fonctions : les arguments sont soit des **int** soit des pointeurs. Les structures ne peuvent pas être passées par valeur, seulement par adresse. Idem pour les valeurs de retour : les types de retour ne peuvent être que type de base ou pointeur sur structure ou constantes ;
  - pointeurs : les pointeurs autorisés référencent des structures, des types de base ou des fonctions. Les pointeurs de pointeurs ne sont pas autorisés.
- Types des variables et des champs des structures : **int** et pointeur sur structure, sur fonction ou sur int. Il est important de noter que les structures ne peuvent être manipulées que par le biais de pointeurs, et ne peuvent être allouées que par une fonction de type **malloc**. Cette contrainte n'est pas imposée par la grammaire mais est imposée par la sémantique du langage.
- Constantes : elles sont entières et exprimées en base 10.
- Déclaration de variables et fonctions externes : une fonction ou variable peut être déclarée **extern**. Dans ce cas, elle est déclarée mais pas définie dans le programme (pour une fonction, cela veut dire que son code est écrit dans un autre fichier). Cela permet de faire référence à des fonctions écrites en C, ou bien des fonctions de bibliothèques tout en permettant de vérifier les types de toutes les fonctions du programme.
- Instructions : les instructions possibles sont
  - Les expressions. L'affectation est une expression qui retourne la valeur affectée. Les priorités entre opérateurs sont fixées par la grammaire ;
  - les structures de contrôle :
    - la structure **if**.. et **if..else**,
    - la structure **for**,
    - la structure **while** ;
  - **return**. Elle peut retourner une valeur ou non.
  - un bloc d'instructions, composé d'une liste éventuelle de déclarations de variables locales suivie d'une liste éventuelle d'instructions ;
- Conditions : les conditions comparent deux expressions quelconques.
  - les six opérateurs de comparaison dont on dispose sont : $>$ $<$ $<=$ $>=$ $==$ **!=**
  - les deux opérateurs booléens sont **&&** ||
- Les expressions
  - Les quatre opérateurs binaires dont on dispose sont : + **-** ***** /
  - Les quatre opérateurs unaires sont : **-** **&** **->champ** *****. & retourne l'adresse d'une variable, ->champ retourne le champ pointé par un pointeur et * déréférence un pointeur. De plus, la fonction prédéfinie **sizeof** retourne la taille en octets de la structure pointée par la va-

riable passée en argument (c'est le compilateur qui calcule la taille d'un type ou d'une structure) ;
- Le type des expressions est quelconque, les calculs sur les types sont les suivants :
  - Toutes les opérations binaires sont autorisées sur des **int**
  - Seules les opérations binaires suivantes sont autorisées sur des pointeurs : l'addition ou soustraction d'un entier à un pointeur (donne un pointeur), la soustraction entre deux pointeurs (donne un entier).
  - Pour les opérations unaires, ***** et **->champ** ne peuvent s'appliquer qu'à une variable de type pointeur, **&** ne peut s'appliquer qu'à une variable de type int ou à une fonction, **-** ne peut s'appliquer qu'à une variable de type **int**.
- Les commentaires débutent par /***** et se terminent par *****/

3.2 Exemple d'un programme écrit en STRUCIT-frontend

Le code `exemple-strucit-frontend.c` est fourni en exemple. Vous pouvez tester que gcc le compile correctement avec la commande `gcc -c exemple-strucit-frontend.c`, cela produira un code objet malgré l'avertissement généré par gcc.

3.3 Grammaire du langage STRUCIT-frontend

La grammaire du langage STRUCIT-frontend est donnée avec une description yacc de départ dans le fichier appelé `structfe.y`. Vous pouvez modifier ou compléter cette description yacc pour le besoin de votre projet. Mais attention, il faut que votre fichier yacc ne génère aucun conflit *shift-reduce*.

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 STRUCIT-frontend ; par exemple, les mots clés du langage C qui n'appartiennent pas au langage STRUCIT-frontend doivent être enlevés, etc. Il est à rappeler que l'analyseur lexical ne doit renvoyer que les tokens valides vers l'analyseur syntaxique, tout autre token qui n'appartient pas au langage considéré doit être négligé.

4 Langage destination : STRUCIT-backend

4.1 Description du langage STRUCIT-backend

Le langage destination STRUCIT-backend est un dialecte du C, proche de l'assembleur. Les types permis pour les variables sont **int** et **void ***, les types construits avec **struct** n'apparaissent pas dans ce langage. De plus, les instructions doivent correspondre à du code trois adresses. Un code trois 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 STRUCTIT-backend n'est pas conforme à toutes les normes C actuelles, ainsi un compilateur C peut rejeter un programme écrit en STRUCTIT-backend. Une description plus complète est donnée ci-dessous.

- Les identificateurs :
  - ils doivent commencer par une lettre, et ne comportent que des lettres, chiffres et soulignés ;
  - ils ne peuvent pas porter le nom d'un mot clé réservé : **extern, int, void, goto, if, return**.
  - Types apparaissant dans le programme : les types autorisés sont **int**, **void** et **void ***. Pour les fonctions, les arguments sont soit des **int** soit des **void ***.
  - Types des variables : **int** et **void ***.
  - Constantes : elles sont entières et exprimées en base 10.
  - Déclaration de variables et fonctions externes : une fonction ou variable peut être déclarée **extern**. Dans ce cas, elle est déclarée mais pas définie dans le programme (pour une fonction, son code est dans un autre fichier). Cela permet de faire référence à des fonctions écrites en C, ou bien des fonctions de bibliothèques tout en permettant de vérifier les types de toutes les fonctions du programme.
  - Etiquettes (*labels*) : les étiquettes (*labels*) apparaissent devant les instructions. Ils sont sous la forme **etiquette :**. Les noms des labels doivent respecter les mêmes règles que les identificateurs. Ils sont locaux à une définition de fonction.
  - Instructions : les instructions possibles sont :
    - Les expressions. L'affectation est une expression. Les priorités entre opérateurs sont fixées par la grammaire. Les affectations ne peuvent comporter au plus qu'un seul opérateur, en plus de l'affectation. Les affectations autorisées sont :
      - **x=y**
      - ***x=y**
      - **x=y op z**, avec **op** un opérateur binaire,
      - **x=op y**, avec **op** un opérateur unaire,
      - **x = y(..)** ou chaque paramètre de l'appel de la fonction y est une constante ou une variable.
    - Les instructions de contrôle :
      - le branchement conditionnel **if .. goto L**. Si la condition après le **if** est vérifiée, alors le contrôle va à l'instruction étiquetée par le label **L** ;
      - le branchement inconditionnel **goto L**. Branchement à l'instruction étiquetée par **L** ;
      - L'instruction **return**. Elle prend ou non une valeur constante ou variable en paramètre.
- Conditions : les conditions comparent deux expressions quelconques. Les six opérateurs de comparaison dont on dispose sont : > < <= >= == **!=**. Les "ou/et logique" entre conditions ne sont pas autorisés (ils

doivent être scindés en plusieurs if).
- Les expressions :
  - Les quatre opérateurs binaires dont on dispose sont : + **-** ***** /
  - Les trois opérateurs unaires sont **-** **&** *****. & retourne l'adresse d'une variable et * déréférence un pointeur. Il n'y a pas d'opérateur -> car le type structure n'existe pas.
- Les commentaires débutent par /***** et se terminent par *****/

4.2 Exemple d'un programme écrit en STRUCIT-backend

Le code `exemple-strucit-backend.c` est fourni en exemple. Vous pouvez tester si gcc le compile correctement avec la commande `gcc -c exemple-strucit-backend.c`, cela produirait des erreurs. Dans le passé, un tel programme était compilable avec d'anciennes générations de compilateurs C !

4.3 Grammaire du langage STRUCIT-backend

La grammaire du langage STRUCIT-backend définie avec une description yacc dans le fichier appelé `structbe.y`. Vous pouvez modifier ou compléter cette description yacc pour votre besoin.

Pour une description lex du langage STRUCIT-backend, 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 STRUCIT-backend. Les descriptions lex et yacc du langage STRUCIT-backend sont nécessaires pour réaliser un parseur pour le langage STRUCIT-backend. Ce parseur permet de vérifier que votre code généré est correct lexicalement et syntaxiquement. La vérification sémantique du code généré se fera à la main simplement.

5 Conseils pratiques

Un projet de compilation est très important dans la vie d'un étudiant en science 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.

Concernant le travail par binôme, nous ne voulons pas qu'un des deux membres fasse tout le projet pour l'autre (nous avons été des étudiants comme vous, nous savons comment cela se passe dans les projets). 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 ou compliqué. C'est vrai, c'est pour cette raison que cela vaut le coup de le faire. 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.
- Étudiez les manuels de lex et yacc. Maîtrisez d'abord ces outils.
- 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. Les tokens qui n'appartiennent pas au langage doivent être négligés à ce niveau, et ne pas être renvoyés à l'analyseur syntaxique par la suite.
- Une fois que vous êtes sûrs de votre analyseur lexical, vérifiez que votre parseur (analyseur lexico-syntaxique) fonctionne correctement sur tous les programmes tests. Vérifiez sa robustesse en modifiant les programmes tests pour provoquer des erreurs syntaxiques, absence de conflits *shift-reduce*, etc.
- 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.
- Si vous estimez que la grammaire yacc fournie a besoin d'être modifiée, faites le, et indiquez lors de la soutenance pourquoi avez vous modifié la grammaire.
- Attention, en langage C, contrairement à d'autres langages, les chaines de caractères doivent être allouées en mémoire explicitement. Si une routine sémantique de yacc accède à une chaine de caractères non allouée en mémoire, votre logiciel plantera.
- Un programme C doit libérer toutes les structures de données allouées dynamiquement. Sinon, si un programme C se termine sans libérer toutes ses structures de données dynamiques, on dit qu'il contient un *memory leak*, que certains validateurs ou analyseurs de codes détecteraient. Dans des entreprises rigoureuses de développement logiciel, il est interdit d'avoir des *memory leaks*. Bien que nous n'exigeons pas cela pour le projet de compilation, entrainez vous quand même à écrire un code C sans *memory leak* si possible avec un outil comme `valgrind`.

Je vous souhaite tout le succès.
Cordialement, Pr Sid Touati