Projet : A2C

Nom du groupe : Ghom

Membres :
Thibaud “zehir” Michaud (michau_n)
Lucien “luciano” Boillod (boillo_l)
Charles “arys” Yaiche (yaiche_c)
Maxime “kylox” Gaudron (gaudro_m)

11 janvier 2015

Présentation du projet

Description

Notre projet est un compilateur de langage algo (enseigné en classe préparatoire à l'EPITA). Il ne traduira pas le langage algo vers une représentation intermédiaire classique telle que llvm ni vers du bytecode pour une machine virtuelle, comme la plupart des compilateurs, mais vers une représentation de plus haut niveau : le langage C. Cela nous permettra de nous concentrer sur le front-end (lexing, parsing, analyse sémantique, sucre syntaxique, ...) sans se soucier des détails de bas niveau, tout en permettant une compilation indépendante de la plateforme. Nous n'allons pas nous servir d'outils de génération d'analyseur lexical et de parseur mais écrire ces deux phases à la main. A terme le projet pourrait servir aux SPE qui utilisent ces deux langages au quotidien.

Le langage

Notre intention n'est pas de respecter à la lettre près les spécifications du langage. Nous allons d'abord essayer de couvrir un sous-ensemble de celui-ci contenant le strict nécessaire : assignation de variables, structures de contrôle, typage, ...

Nous espérons tout de même, à terme, obtenir un langage complet et qui puisse trouver son utilité auprès des étudiants.

Le compilateur

Front-end

Un compilateur ne faisant que de la traduction de langage et de l'analyse sémantique serait très limité. Nous projetons donc d'y intégrer les fonctionnalités suivantes :

- Renvoyer des messages d'erreur clairs en cas d'erreur syntaxique ou sémantique dans le code
- Gestion plus ou moins fine d'avertissements (à l'image des flags -Wall, -Werror, -Wpedantic, etc. des compilateurs C classiques)
- Générer des fichiers représentant les différentes phases de la compilation : visualisation de l'arbre syntaxique, génération du fichier C, du code assembleur,...
- De manière générale, certains flags du compilateur C utilisé pour le back-end seront intégrables sans efforts supplémentaires : niveaux

d'optimisation, génération de code "PIC" (position-independant code), options relatives à la plateforme,...

Back-end

Nous avons décidé de ne pas créer un compilateur complet car cela aurait représenté une masse de travail trop importante pour notre niveau et nos connaissances encore limitées dans ce domaine. Nous avons donc choisi de nous concentrer sur la partie qui nous semblait la plus intéressante : le front-end. Étant donné que le front-end génère du code C, le back-end sera donc un compilateur C classique. Les choix évidents sont gcc et clang, mais à priori rien ne nous empêchera de laisser un choix complètement libre à l'utilisateur.

Intérêt algorithmique

Un compilateur est un projet aussi exigeant qu'intéressant algorithmiquement. En effet cela demande une énorme base théorique, une rigueur de travail et une connaissance générale d'algorithmie. L'implémentation de structures tels que les arbes, les listes, les tables de hachages, etc. seront monnaie courante, il est nécessaire d'être familier avec ces notions afin de se concentrer rapidement sur "la partie théorique". Cette dernière se composera principalement de théorie des langages : analyse lexicale, parseur, analyse sémantique, etc.

Construction du langage algo

Mise en place d'une grammaire (résolution des ambiguités etc), propre à notre langage, puisque nous visons dans un premier temps un sous ensemble du langage algo que nous personnaliseront à notre convenance. Cette partie fera principalement appel aux notions de THL, des méthodes de construction et résolution de grammaire et d'établissements des règles afin de fournir un langage stable, clair et facile à manipuler.

Les différentes phases du front-end

— Lexing : La manipulation d'expressions régulières sera un point important dans cette phase, puisqu'elle permettra de découper le programme en "tokens", c'est à dire en groupes de caractères représentant une unité grammaticale.

Algorithimiquement il s'agira ici d'avoir des structures de recherche et de reconnaissance de nos tokens, éventuellement à partir d'automates.
- Parsing : Nous avons le choix parmis les parseurs les plus courants :
  - LL(1)
  - LR(1)
  - LALR(1)
  - SLR(1)
  - descente récursive, plus simple à implémenter mais moins performante

Mais il s'agira surtout du sélectionner la méthode la plus perfomante et adaptée à notre grammaire, pour cela nous pourrons créer l'automate associé pour faciliter la résolution de la grammaire ainsi que la construction du parseur.

La phase de parsing présente un intêret algorithmique important puisque l'efficacité de notre compilateur résultera directement de notre parseur.

Analyse sémantique

Nous allons ici devoir implémenter une structure de donnée efficace de "dictionnaire" (table de hashage, arbre auto-équilibré) pour implémenter la table des symboles, permettant d'associer à chaque identifiant des informations sur son type, sa portée, etc.

Génération de code

La génération de code consistera à utiliser l'arbre syntaxique obtenu à partir des phases précédentes pour génerer un fichier source en C. Pour cela, chaque noeud de l'arbre syntaxique aura une fonction qui lui sera associée et dont le rôle sera de génerer le code C adéquat.

Nos sources

- Le projet Tiger Compiler d'EPITA
  (https ://www.lrde.epita.fr/ akim/ccmp/assignments.html)
- Modern compiler implementation in ML
- Compilers : Principles, Techniques, and Tools (Dragon Book)
- http ://llvm.org/docs/tutorial/
- http ://www.stack.nl/ marcov/compiler.pdf