Cahier des Charges

A2C

Team Ghom
Promo 2018

Thibaud "zehir" Michaud
Lucien "Luciano" Boillod
Charles "Arys" Yaiche
Maxime "Kylox" Gaudro

10 Janvier 2014

Table des matières

Introduction

A2C est un projet informatique réalisé dans le cadre de la seconde année des classes préparatoires de l'EPITA, école d'ingénieur en informatique. Le projet s'étale sur 4 mois et le sujet est libre. C'est pourquoi nous avons décidé de nous lancer dans la réalisation d'un compilateur du langage algo (langage 'théorique' proche du pascal, utilisé à but pédagogique à EPITA) vers le langage C.

Chapitre 1

Présentation du projet

Le but de ce projet est d'écrire un compilateur à partir de zero. Il compilera le langage algorithmique, enseigné en classe préparatoire de l'EPITA, vers du C.

Autrement dit, nous n'allons implémenter que la partie "front-end" du compilateur : analyse lexicale, syntaxique et sémantique. La partie "back-end" sera gérée par un compilateur C tel que gcc ou clang.

Ce choix est dû au fait que l'écriture complète d'un compilateur nous semble trop ambitieux, et que la partie "front-end" est celle qui nous paraissait la plus intéressante d'un point de vue algorithmique.

Le public cible de ce compilateur sont les sups et les spés utilisant le langage algo en cours et en travaux pratiques. Il peut également servir à toute autre personne, car il sera libre et open source.

1.1 Presentation langage algo et equivalence C

Le langage algo est un langage inventé dans le but de faire comprendre les cours d'algo aux élèves de prépa à l'EPITA, ce qui n'est pas chose aisée. Ce langage a donc intérêt à offrir moins de souplesse que le C et à bien faire la distinction entre les différents types, les procédures les fonctions, etc. Nous allons donc faire un bref aperçu de la structure générale de ce langage.

1.1.1 structure de base

de base le langage se structure de la sorte :

```
[procedure ou fonction] algorithme [nom algo]
  [declaration des variables et des parametres]
debut
  [implementation]
fin algorithme [nom algo]
```

qui pourrait alors se traduire en C par :

```
[type de retour] [nom algo]{[definition des parametres]){
  [declaration des variables]
  [implementation]
}
```

On voit déjà apparaître que le langage algo est plus éxigeant en terme de mots clefs que le C. Passons maintenant à la déclaration des variables et paramètres.

1.1.2 Déclaration des variables et des paramètres

Parmis les déclarations des variables et des paramètres il existe plusieurs déclarations possible suivant la composante algorithmique utilisée.

```
  constantes
    id_const = valeur
  types
    declaration de type
  variables
    id_type id_var 1
```

qui pourrait alors se traduire en C par :
- pour les constantes :

```
    const [type] id = valeur;
```

- pour les types :

```
   typedef [type] id = [type];
```

- pour les variables :

```
    [type] id = valeur;
```

1.1.3 Structure de choix

Passons maintenant à la presentation des structures de choix et à leurs équivalences en C. La structure est un peu verbeuse mais sa traduction vers le C ne pose pose pas de problème majeur.
Algo :

```
  si [condition] alors
    [instruction 1]
  sinon
    [instruction 2]
  fin si
```

C :

```
  if([condition]){
    [instruction 1]
  }
  else{
    [instruction 2]
  }
```

1.1.4 Les boucles

Comme en C, en algo il existe deux types de boucles : `tant que` et `pour`, respectivement `while` et `for` en C. Les deux boucles sont là aussi sensiblement équivalentes dans les deux langages, leurs syntaxes sont : en algo

```
tant que [condition] faire
  [instruction]
fin tant que
```

en C

```
while([condition]){

}
```

Il existe aussi la boucle `faire tant que` qui est l'équivalent de la boucle `do while` en C : en Algo

```
faire
  [instruction]
tant que [condition]
```

en C

```
do{

}while([condition])
```

Et finalement la boucle `pour` : en algo

```
pour i <- [valeur 1] jusqu a [valeur 2] faire
  [instruction]
fin pour
```

en C

```
for([initialisation];[existance];[incrementation]){
  [instruction]
}
```

Voilà les bases de la syntaxe du langage algo. Il nous est alors évident que le passage du langage algo vers un langage plus courant comme le C nous semble facile, on arrive assez facilement à trouver les correspondances entre les différentes syntaxes. C'est pour cela que nous avons choisis le C, pour sa syntaxe simple et par sa grande utilisation.

l'ensemble de document de cour sur le langage peuvent etre trouver sur le site (si il est à jour) de junior : http ://algo-td.infoprepa.epita.fr/algo/langage/

Chapitre 2

Présentation du groupe

2.1 Présentation de l'équipe

2.1.1 Lucien "Luciano" Boillod

Je me suis toujours intéressé aux nouvelles technologies et tout ce qui touche à l'informatique. Je suis quelqu'un d'investi et ambitieux dans tout ce que j'entreprends, seul ou en groupe. Durant mon temps libre je tiens un blog ainsi qu'un ensemble de sites web, où je poste des articles en rapport avec l'informatique. J'ai également contribué au projet Vcsn du LRDE pendant quelques mois en tant que stagiaire, ce qui m'a beaucoup inspiré et motivé à donner le meilleur de moi-même.

2.1.2 Maxime “Kylox” Gaudron

Seul représentant chevelu du groupe, suite aux mésaventures de sa destination à l’étranger entre complot illuminati et tactique de franc-maçon il a fallu refonder non pas une famille mais un groupe de projet pour démarrer ce S4 de la plus belle manière. c’est alors que proposant de travailler sur un kernel à zéphir, il lui répondit "non j’ai pas envie, viens on fait un compilateur" ne sachant que dire face à une telle détermination il accepta sans soucis, de toute manière les deux sont tout aussi intéressants. Il a alors fallu trouver d’autres gens parce que deux c’est sympas mais quatre c’est mieux ! Après avoir feuilleté le résultat des contrôles d’IP de leur nouvelle promos ils se rendirent alors compte qu’ils ne connaissaient pas grand monde ... problématique ... Lors d’un bref passage au LRDE zéphir rencontra la personne qui allait devenir notre responsable technologie annexe (a.k.a Apple) Luciano. Et ce fut à mon tour de trouver le dernier membre de notre compagnie, ayant en tête la bonne capacité de notre groupe il nous fallait quelqu’un de différent, c’est pour cela que notre choix s’est porté sur Arys le saltimbanque ! Bon élément, aimant développer et toujours souriant, il se lia alors au groupe . Pour ma part je trouve que ce groupe peut faire quelque chose d’intéressant ensemble si notre chemin n’est pas corrompu par les siestes interminables et les bouteilles de bière à finir.

2.1.3 Charles “Arys” Yaiche

Un bel homme originaire à la fois de la Tunisie, de la Pologne, de l'Italie et du Pays basque, Arys (de son nom d'origine Charles mais comme c'était pas assez swag Arys c'est mieux, et puis c'est un peu une marque de pain brioché, et ça c'est la classe) est né dans la ville rose, j'ai nommé, Toulouse... on ne peut pas être parfait. Ce valeureux personnage a très rapidement fui Toulouse avant même de savoir parler (pour ne pas choper l'accent) pour aller se réfugier dans la capitale, c'est dans cette même capitale qu'il fera une partie de sa vie. Notre personnage agé maintenant de 16 ans décide de quitter la maison et de partir à l'aventure à l'autre bout du monde, j'ai appellé Chicago pendant deux ans, deux ans plus tard, il passe avec succès le Baccalauréat mention "raz des fesses" et est accepté dans "la meilleure école d'ingénieur en informatique de France" citation du directeur. Il valide sa première année, il va... redoubler sa seconde année lors de son départ en Chine parce que un chinois l'avait entubé lui et ses camarades, mais il faut savoir que tout est de notre faute car de toute façon tout est toujours notre faute et il parait que on a rien foutu aussi d'après la direction qui est en contact permanant avec l'université de Chine. C'est donc lors de ce quatrième semestre avec une promotion inconnue que notre héro doit s'intégrer pour former un groupe de projet libre, je trouve que il s'est très bien intégré sachant que l'ensemble des membres du groupes sont tous redoublant.

2.1.4 Thibaud “Zehir” Michaud

C'est avec les calculatrices scientifiques utilisées au lycée que j'ai commencé à m'intéresser à la programmation. Alliant à la fois la logique implacable des mathématiques et le côté pratique de la physique, l'informatique était pour moi la science qui manquait dans l'enseignement. C'est alors que j'ai découvert, ô joie, que certaines écoles étaient (presque) entièrement dédiées à ce domaine. L'EPITA en faisait partie et proposait en plus une pédagogie par projet particulièrement attrayante.

Je suis de nature ambitieuse, et ça m'a d'ailleurs coûté un semestre. Parti pour un semestre d'étude à Tampere (Finlande), destination réputée difficile, j'ai voulu prendre des cours qui semblaient intéressants mais pour lesquels il s'est avéré que je n'avais pas le niveau.

Me voilà donc avec un semestre de libre (puisque déjà validé l'année précédente) durant lequel je fais un stage au Laboratoire de Recherche et de Développement de l'EPITA (LRDE). Durant ce semestre je dois trouver un sujet et un groupe de projet pour le semestre suivant.

Pour ce projet de 4ème semestre, je voulais quelque chose d'ambitieux mais pas trop. La suggestion de Maxime, faire un kernel, me paraissait pour le coup un peu trop ambitieuse. Mais on a fini par tous se mettre d'accord sur la réalisation d'un compilateur.

2.2 Philosophie du groupe

En tant qu'étudiants en infornatique, les nouvelles technologies nous passionnent, très rapidement immergés dans le monde Unix, nous sommes militants pour le logiciel libre (à l'exception de Lucien qui utilise un Mac). Fervents utilisateurs de vim (sauf Lucien qui utilise emacs) ergonomie est notre mot d'ordre

avec deux membres utilisant des claviers ergonomique bépo et dvorak.

Partant du principe que nous sommes là pour apprendre, nous avons décidé de tout implémenter nous même : structures de données, parseur, etc. (à l'exception du back-end parce que faut pas exagérer quand même).

Chapitre 3

Découpage du projet

Les premiers compilateurs ne faisaient qu'une seule passe. C'est à dire qu'ils généraient le code machine au fur et à mesure qu'ils parcouraient le code source. Cette approche à plusieurs avantages mais a surtout un gros désavantage : elle n'est pas modulaire et est difficile à maintenir.

Les compilateurs modernes préfèrent donc effectuer plusieurs passes : analyse lexicale, analyse syntaxique, analyse sémantique et génération de code machine. Certaines de ces étapes peuvent encore être subdivisées, mais c'est ce découpage global que nous allons suivre pour notre projet, et que nous allons détailler dans cette partie.

Mais avant de se lancer dans la programmation du compilateur, il est important que nous ayons une implémantation fiable et facile d'utilisation de quelques structures de données de base.

3.1 Structures de données

> Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> Linus Torvalds

La plupart des langages de programmation haut niveau fournissent des structures de données génériques dans leur bibliothèque standard, car les structures de données sont au cœur de la programmation. Mais notre projet doit être fait en C, et en C la bibliothèque standard ne contient pas de structures de données.

Il semblerait qu'il existe des bibliothèques non standard implémentant des structures de données génériques, telle que klib, mais on préfère tout réimplémenter nous même, c'est une bonne occasion de mettre en pratique tout ce qu'on a vu en algo. [cf. philosophie du groupe]

Un gros obstacle sera la généricité. Les différentes étapes de la compilation vont utiliser les mêmes structures de données, mais avec des données de types différents. Et le C étant relativement bas niveau, rien ne permet d'avoir nativement des types génériques.

Pour palier à ce probléme plusieurs solutions existent parmis lesquelles :

— Implémenter les structures sous forme de macros. Les macros prennent en paramètre le type utilisé et seront remplacées à la compilation par le code implémentant les structures de données avec le bon type. C'est un peu le même principe que les templates C++ en plus basique. C'est aussi la solution adoptée par la bibliothèque klib (https ://github.com/attractivechaos/klib).
— Utiliser des types `void *` dans toutes les structures, et caster en fonction du type utilisé. Cette solution est probablement la plus simple, mais on perd tout l'intérêt de l'analyse de type statique.
— Utiliser des structures de données intrusives : le principe est d'inclure la structure de données dans les données plutôt que l'inverse. C'est notamment la solution adoptée par le noyau Linux.

Chacune de ces méthodes à des avantages et des inconvénients. Mais la dernière solution semble être celle qui offre le meilleur compromis, et c'est probablement celle que nous allons choisir.

Une fois les structures de base implémentées (listes chainées, tables de hachage, etc.) nous allons pouvoir attaquer l'implémentation de la toute première étape de la compilation : l'analyse lexicale.

3.2 Analyse lexicale

L'analyse lexicale consiste à découper le code source en entités lexicales, ou lexèmes ("token" en anglais), c'est à dire transformer le code source en un flux de mots-clés, d'opérateurs, d'identifiants, de valeurs numériques, ...

3.3 Analyse syntaxique

L'analyse syntaxique consiste à trouver la structure grammaticale du code source à partir du flux de tokens renvoyé par l'analyseur lexical. Dans tous les compilateurs modernes, cette structure est exprimée sous la forme d'un arbre dit "Arbre syntaxique abstrait", ou AST.

3.3.1 Construction de la grammaire

Qu'est-ce qu'une grammaire ?

Dans les langages naturels, une grammaire construit les phrases, elle ne s'intéresse pas au sens, mais à la syntaxe. Dans les langages informatique, cette grammaire agit pareil, elle va participer à la correction syntaxique, et non au sens des expressions formées. Ainsi une phrase comme "Elle mange une voiture" est syntaxiquement correcte, tout comme "int positive = -1 ; " l'est.

Une grammaire formelle (ou, simplement, grammaire) est constituée des quatre objets suivants :

- Un ensemble fini de symboles, appelés symboles terminaux (qui sont les « lettres » du langage), notés conventionnellement par des minuscules.
- Un ensemble fini de symboles, appelés non-terminaux, notés conventionnellement par des majuscules.
- Un élément de l'ensemble des non-terminaux, appelé axiome, noté conventionnellement S.
- Un ensemble de règles de production, qui sont des paires formées d'un non-terminal et d'une suite de terminaux et de non-terminaux.

Ainsi une grammaire peut être caricaturée comme un ensemble de règles auquel obéissent des variables, le but étant de caractériser le langage.

Quel type de grammaire ?

Il existe plusieurs types de grammaires (monotones, sensibles au contexte, rationnelles ... etc) définis par la hierarchie de Chomsky. Il s'agira de choisir une grammaire conformément à notre langage, ainsi l'idée d'une grammaire finie ou rationnelle est tout bonnement proscrite.

En effet la construction d'un langage informatique, ne peut se faire via quelque chose de rationnel. L'une des raison à cela est le problème soulevé par anbn, qui peut représenter le matching de toutes les parenthèses, que la rationnalité ne peut détecter.

Il reste donc à voir si une grammaire Hors Contexte (CF pour context free en anglais) peut suffire, ou si on doit extrapoler à une grammaire sensible au contexte. La réponse est oui, la principale barrière induite par la rationnalité

étant détruite, le Hors Contexte suffit à la réalisation de la plupart des langages informatique, dont le notre.

Definition : Une grammaire hors contexte est tout simplement une grammaire formelle dans laquelle chaque règle de production (ou simplement production) est de la forme X -> w où X est un symbole non terminal et w est une chaîne composée de terminaux et/ou de non-terminaux.

Realisation

La construction de la grammaire peut être separé en deux étapes majeures. La première étant de construire la grammaire qui définie notre langage, sans s'occuper des quelconques problèmes (cette grammaire est souvent appelée grammaire abstraite). La seconde étant la résolution de cette grammaire, c'est à dire résoudre les divers conflits et ambiguités (souvent appelée grammaire concrête). Une ambiguité est tout simplement deux chemins possibles. Il est également possible de créer un automate à partir de la grammaire, ce qui peut grandement simplifier la réalisation du parseur.

3.3.2 Ecriture de l'analyseur syntaxique

La construction d'un parseur sera une partie assez longue et fastidieuse. Cependant elle sera grandement facilité par l'étape précédente, à savoir la construction de la grammaire. Il existe deux grandes familles de parsing : LR(1) et LL(1). Le chiffre correspong au nombre de 'look ahead' (symbole de surete pour regarder en avant dans le parsing). La différence entre LR et LL provient de la dérivation sur laquelle on travaille : gauche (left) pour LL et droite (right) pour LR.

Plusieurs choix s'offrent donc à nous, il s'agira ici de choisir entre performance et faisabilité. Si la grammaire est LL, il sera alors beaucoup plus simple d'implémenter un parseur LL. Si au contraire la grammaire est LR, il faudra se lancer dans un parseur LR, plus compliqué mais très efficace et auquel pour se rajouter des amélioration comme LR(0), LALR(1) ou SLR(1).

3.4 Analyse semantique

L'analyse sémantique consiste à verifier l'exactitude des types, des déclarations,... . Elle a pour but de verifier le sens de la phrase ecrite.

```
1              si Vrai == 0 alors
2                  Vrai + 1
3              fin si
```

Cette phrase est syntaxiquement juste mais sémentiquement fausse

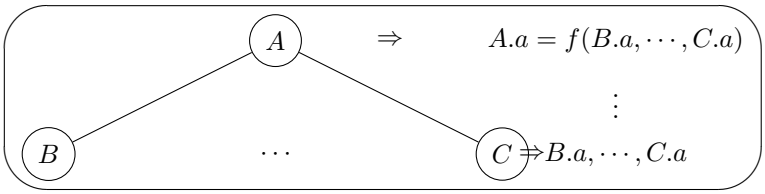
Pour palier a ce problème on va mettre un place un système d'attribut, qui donnera un sens, une valeur aux outils de la grammaire avec lesquels nous avons réalisé l'analyse syntaxique ce sens sera definis par les attributs.

3.4.1 Les attributs

L'analyse sémantique repose sur un principe : Un attribut est une valeur quantifiée associée à un élément du langage. L'élément peut être n'importe quoi, un type d'expression , une valeur, etc etc . Chaque élément du langage est représenté par un symbole auquel l'attribut sera associé. on note alors, A.a ou a est l'attribut de l'élément A. Le calcule de ces attributs se fait en fonction de la grammaire utilisée dans l'analyse syntaxique on associes donc à chaque aspects de la grammaire des règles sémentiques. cette grammaire est alors dites attribuée et l'arbre syntaxique qui porte alors en chacun de ses noeud la valeurs des attributs du même noeud sera appelés arbre syntaxique décoré. Il existe deux types d'attributs.

Attributs synthétisé

Un attributs synthétisé se calcule au niveau d'un noeud A en fonction de la valeur d'attribut de ses fils



Il s'agit donc d'un analyse ascendante.

Attributs hérités

Ces attributs sont un peu l'inverse des attributs synthétisés car ils ne sont pas calculé a l'aide des fils mais à l'aide des noeuds parents et des noeuds frères. Cette analyse est qualifié de descendante et permet de situé chaque noeud dans un contexte.
Une fois ces calculs effectués notre analyse va pouvoir verifier si la phrase a un sens en fonction des attributs calculés. Si le resultats en tout noeud de

notre arbre est logique (reviens basiquement a dire que 1+1 = 2) alors notre sémantique et respecter et nous pouvons passer a la génération de code.

3.4.2 Ouverture de l'analyse sémantique

L'analyse sémantique peut devenir vraiment complexe si l'on decide de traiter plusieurs sources et plusieurs destinations. C'est a ce moment qu'intervient les schemas de compilation qui permettront de definir plusieurs analyses differente mais pour le moment nous resterons uniquement sur une compilation d'une source vers une destination.

3.5 Génération de code C et compilation du code généré

L'arbre syntaxique précédament construit permettera ensuite de générer du code une fois le compilateur ayant générer le code C, celui-ci sera compiler en par n'importe quel compilateur du langage C comme gcc, ou alors clang

3.6 Site web

Comme dans tout projet informatique, la construction d'un site web, et l'écriture d'articles tout le long de la réalisation du projet, est quelque chose d'indispensable à sa communication. En dehors du fait de laisser notre trace sur internet, le site web peut apparaitre comme un portail de communication entre les developpeurs (nous) et les utilisateurs, ou encore avec d'autres développeurs voulant connaitre un peu plus en détail notre compilateur et pourquoi pas nous contacter. De plus le site web permet un téléchargement simple du compilateur et centralise toutes les ressources nécéssaires (github, page FB, liens vers des articles en rapport .. etc).

Le site sera également un élément essentiel dans la réalisation éventuelle d'un de nos bonus, car il hébergera potentiellement le compilateur interactif.

Les technologies utilisées seront HTML/CSS, et Javascript si l'on parvient à mettre en place le bonus.

Chapitre 4

Organisation projet

	première soutenance	deuxième soutenance	troisième soutenance
site web	done	done	done
structure de donnés	done	done	done
lexer	done	done	done
construction gammaire	75%	done	done
parseur	75%	done	done
analyse sémantique	0%	25%	done
génération de code	0%	0%	done

Chapitre 5

Bonus

Dans le cas ou le compilateur serait fini en avance, l'équipe a fait une sélection de bonus.

- Optimisation du code généré
- Gestion de plusieurs fichiers pour la compilation
- Implémentation de bibliothèques pour le langage algo, comme par exemple implémentation des différentes structures de données
- Gestion Français/Anglais pour les classes internationales
- Un compilateur web interactif, ou les élèves pourront compiler leur code en ligne
- Un IDE pour le langage algo
- Un plugin de syntaxe Vim

Bref nous ne sommes pas en panne d'imagination pour toute sorte d'améliorations diverses et variées

Chapitre 6

Technologies utilisées

Langage de programmation	C, HTML, CSS, PHP, Javascript, Python, Shell
outils de versionnement	git, gitlab
Compilateur	clang, gcc
Editeur de texte	Vim, gedit, nano (et emacs...)
OS	Unix
Plateforme de développement continue	jenkins
Analyseur de code statique	sonar
autres (test)	flex, bison

Chapitre 7

Coûts

materiel	cout
serveur	4,99€/mois
T-shirt	40€
lenovo yoga 2	1350€
samsung	700€
samsung	900€
macbook pro(Lucien)	10K€
café	autant que pour le grec et sushi
grec et sushi	autant que le café
peripapeticienne / euphorisants	∞

Conclusion

Nous avons conscience de la charge de travail qu'il faudra fournir au vue de nos compétences peu avancées dans le domaines, puisqu'il nous manque une grosse partie de la base théorique enseignée en ING1. Mais sont sommes extrémement motivés et confiant quant à la réalisation de ce projet, et espérons y prendre du plaisir.

Sources

Nos sources :

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

Annexes