PLD-Comp
Développement d'une chaîne complète de compilation

Dans ce projet, nous allons concevoir un compilateur pour un sous-ensemble du langage C (voir sujet ci-dessous) grâce à flex/bison et l'utilisation du langage C++.

1 Le langage

Il s'agit d'un sous-ensemble du langage C avec les éléments suivants :

- Types de données : restriction aux types `char`, `int32_t` et `int64_t`
- Initialisation d'une variable possible lors de sa déclaration
- Tableaux à une dimension
- Fonctions (déclaration et définition), possibilité (et obligation) d'utiliser le type retour `void` pour construire des procédures. La déclaration de fonction est une partie optionnelle du projet. Si vous choisissez de ne pas la traiter, alors vos exemples devront tous avoir une définition des fonctions *avant* leur utilisation.
- Structures de contrôle : `if`, `else`, `while`, `for`
- Structure de blocs grâce à { et }
- Expressions : tous les opérateurs du langage C y compris l'affectation
- Au choix (les deux si vous en avez le temps) :
  - Déclaration de variables dans n'importe quel contexte (pas seulement au début d'une fonction)
  - Possibilité de déclarer des variables globales
- Utilisation des fonctions standard `putchar` et `getchar` pour les entrées-sorties
- Constante caractère (avec la simple quote)
- Les chaînes de caractères pourront être représentées par des tableaux de `char` (pas de constantes chaînes de caractères)
- Un seul fichier source sans pré-processing (pas d'inclusions externes ni macros, toutefois les directives du pré-processeur devront être autorisées mais ignorées afin de garantir que la compilation par un autre compilateur soit possible. Exemple : inclusion de `stdint.h`)

Exemple de programme :

```
#include <stdint.h>

void main(void) {
  int32_t a;
  int32_t b;
  a=0;
  b=6;
  if (a*5+3*b==18)
  {
    putchar('o');
    putchar('k');
  }
}
```

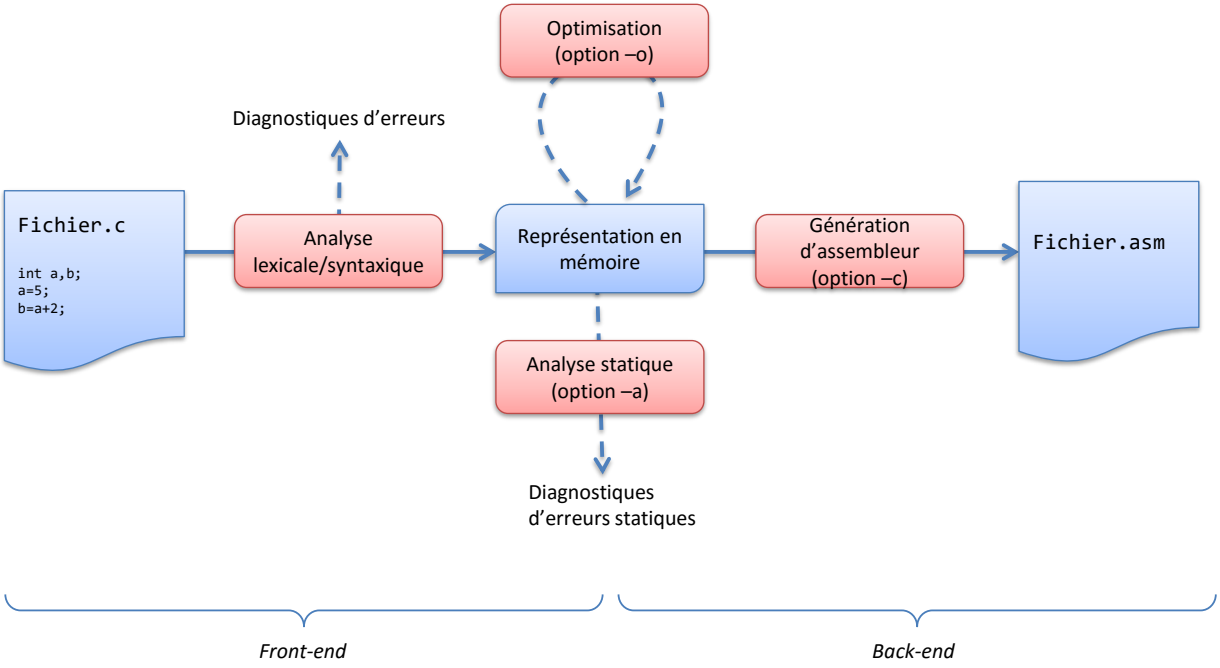
Plus particulièrement le langage qui devra être analysé **ne comprends pas les éléments suivants** :

- Pointeurs (toutefois, on pourra utiliser la notation `[]` comme argument d'une fonction pour passer en paramètre un tableau par son adresse)

— La possibilité de séparer dans des fichiers séparés les déclarations et les définitions
— La structure de contrôle `switch..case`
— La structure de contrôle `do..while`

2 Architecture globale de l'outil

Voici le schéma de l'architecture globale de l'outil :



Le logiciel se présente sous la forme d'un outil en ligne de commande dont l'argument principal est le nom du fichier à analyser. Par défaut, l'outil va analyser le fichier, en faire une représentation en mémoire et afficher des diagnostics d'erreurs (lexicales, syntaxiques ou sémantiques simples). Chaque erreur devra faire apparaître le numéro de ligne et de manière optionnelle le numéro de colonne dans le fichier source. En complément, il existe trois options en ligne de commande qui permettent d'enrichir ce fonctionnement de base :

1. (facultatif) Analyse statique du programme (option -a). Cette option fait une analyse du programme de manière statique afin d'en extraire des erreurs (sur la sortie d'erreur standard).
2. (facultatif) Transformation du programme afin de le simplifier (option -o). En propageant les constantes et en simplifiant certaines expressions (éléments neutres), la représentation interne du programme est modifiée.
3. Génération de code (option -c). Cette option permet de générer le code assembleur dans un fichier séparé.

3 Étapes de travail

Les étapes sont ici données à titre indicatif, organisez vous comme vous le souhaitez. Certaines étapes sont parallélisables, à vous de les détecter. On peut par exemple très bien construire à la main une représentation d'un programme en faisant appel aux constructeurs (donc sans analyse syntaxique ni lexicale). Cela permet de tester certaines fonctionnalités avant l'intégration.

Écriture de la grammaire. Reprenez les éléments du langage exprimés dans la première section et déduisez-en une grammaire. Gardez en vue que vous devrez implémenter cette grammaire dans bison donc qu'il faut le plus possible être adapté à une analyse ascendante (privilégier les récursivités gauches par exemple).

Identification des expressions régulières. Pour chacun des éléments du langage (symboles terminaux), identifiez les expressions régulières associées. Si cela est nécessaire (expressions régulière ayant des éléments communs), mettez des priorités à chacune.

Conception des structures de données. Il s'agit ici de définir les structures qui contiennent le programme lui-même (arbre syntaxique abstrait) et les différents dictionnaires nécessaires (table des symboles, etc).

On demande un diagramme de classes UML.

Vous êtes invités à faire valider cette modélisation par l'équipe enseignante dans le but de ne pas perdre de temps.

Implémentation de l'analyseur avec flex/bison. À partir de la grammaire et des expressions régulières identifiées précédemment, produisez les fichiers flex et bison qui permettent d'analyser le code source afin d'en valider la construction grammaticale. Cette phase n'est qu'une transcription de votre grammaire. Si la grammaire est bien conçue, ce sera une formalité. Malheureusement, vous aurez sûrement des surprises et bison vous indiquera des conflits. Vous devez ou bien résoudre le conflit, ou bien expliquer pourquoi le choix fait par bison lors du conflit est le bon. Pour vous aider à comprendre les conflits, activez l'option `-v` de bison et allez voir le fichier `.output` ainsi généré. N'oubliez pas de paramétrer dans bison les priorités d'opérateurs qui feront disparaître énormément de conflits (directives `%left` et `%right`).

Construction de la représentation en mémoire. Complétez les règles bison par des actions afin de construire une représentation en mémoire du programme. À ce moment du projet, vous pourrez faire la vérification de la conformité de la signature des déclarations et définitions de fonctions (seulement si vous avez choisi d'autoriser les déclarations de fonctions).

Résolution de portée de variables. Dans une expression, lorsqu'une variable est utilisée, on ne connait que son nom. Le but de cette partie est de faire le lien entre les variables utilisées et leurs déclarations. Attention, il y a plusieurs types de variables dont l'emplacement mémoire sera différent au moment de la génération de code assembleur. Par exemple une variable de fonction n'est pas au même endroit qu'une variable globale ou qu'un paramètre de fonction. À l'issue de cette partie, toutes les ambigüités sont levées (par exemple, une variable de fonction qui masque un paramètre de fonction, ou une variable globale). Vous serez à même de détecter dans cette partie l'utilisation de variables non déclarées.

Vérification statique du programme (optionnel). Réfléchissez à l'algorithme qui va permettre de vérifier la validité du programme de manière statique. Voici les éléments qui sont vérifiables dans cette partie :

— Une variable est utilisée (en partie droite d'une affectation ou dans une opération d'écriture) sans avoir jamais été affectée.
— Une variable a été déclarée et jamais affectée ou utilisée.

À la moitié du projet (4 séances), vous devriez avoir terminé toutes les tâches précédentes, et donc avoir le *front-end* opérationnel. Vous pouvez maintenant passer au *back-end*.

Définition des structures de données pour une représentation intermédiaire linéaire. Il s'agit de définir une structure de donnée qui reste abstraite mais se rapproche de l'assembleur. Ce sera un un graphe orienté d'instructions élémentaires à trois adresses (deux variables opérandes, une variable destination). Dans

ce graphe, la plupart des instructions élémentaires n'ont qu'un successeur, seul le saut conditionnel a deux successeurs possibles. Les dictionnaires seront partagés avec l'AST.

Transformation de l'AST vers la représentation linéaire. Le travail ici consiste à "désucrer" le code initial :
- démonter les expressions en séquences de code trois adresse
- démonter les `for`, `while` et `if` en sauts conditionnels
- mettre toutes les variables à plat dans une seule portée

Ceci se fait par un parcours de l'AST. Dans ce processus, on créera autant de variables intermédiaires que nécessaire.

Génération de code assembleur à partir de la représentation linéaire. On produira dans cette partie de l'assembleur, a priori pour un processeur de PC (x86 64-bits), mais l'une des cibles suivantes sera la bienvenue :
- assembleur ARM (par exemple pour Raspberry Pi)
- assembleur MSP430
- java bytecode (sans utiliser le modèle objet)

On demande de l'assembleur sous forme textuelle qui sera assemblé en code machine par `as`. Vous avez donc droit aux *labels* et autres facilités offertes par GNU `as`. Voir en annexe comment ensuite assembler et exécuter votre assembleur sur une machine linux. On ne demande pas d'allocation de registre ici.

Gestion des appels de procédures. Pour les programmes qui ont plusieurs procédures, il faudra gérer la pile des appels, sur laquelle seront placées toutes les variables locales de chaque procédure.

Outil en ligne de commande - intégration. Il s'agit là de concevoir et implémenter l'outil qui va permettre d'interfacer l'analyseur.

Optimisations (facultatif mais recommandé). Certaines optimisations peuvent se faire sur l'AST :
- propagation de constantes : une expression dont toutes les opérandes sont des constantes sera remplacée par la valeur résultante.
- éléments neutres : pour chaque opérateur qui possède un élément neutre (l'addition le 0, la multiplication le 1, *etc.*), les opérations correspondantes peuvent être supprimées lorsqu'elles surviennent.

Certaines optimisations se font sur la représentation linéaire :
- destruction des instructions redondantes (optimisation à fenêtre)
- calcul de vivacité des variables et allocation des registres

Et voici une extension/optimisation qui touche à la fois *front-end* et *back-end* : le support du `switch/case` du langage C, implémenté par une table de sauts. C'est une optimisation car, sur des gros `switch`, la table des sauts va bien plus vite que des `if` imbriqués.

4 Livrables (zéro papier)

Le projet se déroulera en deux temps avec un document de conception à mi-parcours à déposer sur moodle qui reprendra les éléments suivants :
- la grammaire du langage que vous avez utilisée
- la description des structures de données sous forme d'un diagramme de classes
- la liste des fonctionnalités implémentées

À l'issue du projet, un livrable de réalisation sera déposé sur moodle. Ce dossier de réalisation (application testable sur les machines LINUX du département) sera déposé sur moodle sous la forme d'une archive zip [1]. Merci de nettoyer l'archive pour ne pas qu'elle contienne des binaires ni des répertoires cachés. Le zip

1. ce format d'archive permettra d'éviter d'encoder les droits d'accès sur les fichiers comme le ferait une archive tgz

contiendra :

- tous les sources de votre application
- le `makefile` avec une cible par défaut qui crée l'exécutable et une cible « test » qui lance les jeux de tests
- vos jeux de tests

Enfin, une présentation orale montrera l'état de l'avancement du projet et l'exécution commentée des tests dans divers cas à l'issue du projet (pendant les deux dernières heures de la dernière séance).

Remarque : ce sujet sera complété avec des informations supplémentaires sur la partie *back-end*.