Grammaires et Langages - Projet
Développement d'un interpréteur pour le langage lutin

L'écriture d'un compilateur est une tâche complexe et longue. Dans ce projet, nous allons nous focaliser sur certains aspects de la compilation d'un langage de programmation simple, de l'analyse jusqu'à l'interprétation en passant par la simplification sémantique et la détection/récupération d'erreurs.

1 Langage Lutin

Lutin est un langage de programmation algorithmique très simple. Il permet d'effectuer des traitements dans le domaine des valeurs numériques. Les variables ne peuvent être que des variables simples. Les restrictions sont énumérées dans ce qui suit.

Exemple de code Lutin :

```
var x ;
const n=10, n2=100 ;
ecrire n+n2;
x := n+n2 ;
ecrire x+2 ;
lire x ;
ecrire x;
```

Le programme est composé de deux parties, une partie déclarative (déclaration de variables et de constantes) et une partie instructions. Chaque déclaration ou instruction se termine par un point virgule. Les espaces, tabulations ainsi que les sauts de ligne sont facultatifs et servent uniquement à séparer les éléments et introduire plus de lisibilité.

Le mot-clé `var` sert à déclarer une variable. Il est suivi d'une liste d'identificateurs séparés par une virgule.

Le mot-clé `const` sert à déclarer une constante numérique. Il est suivi d'une liste de constantes elles-mêmes consituées d'un identificateur, du symbole égal et d'une constante numérique. Les constantes sont séparées par le symbole virgule.

La partie déclarative peut contenir autant de déclarations de variables et constantes que souhaité et ce dans n'importe quel ordre.

Il existe trois types d'instructions. Le mot-clé `ecrire` permet d'écrire sur la sortie standard le résultat de l'évaluation de son argument (une expression).

Le mot-clé `lire` permet de saisir sur l'entrée standard une valeur numérique et de l'affecter à la variable donnée en argument.

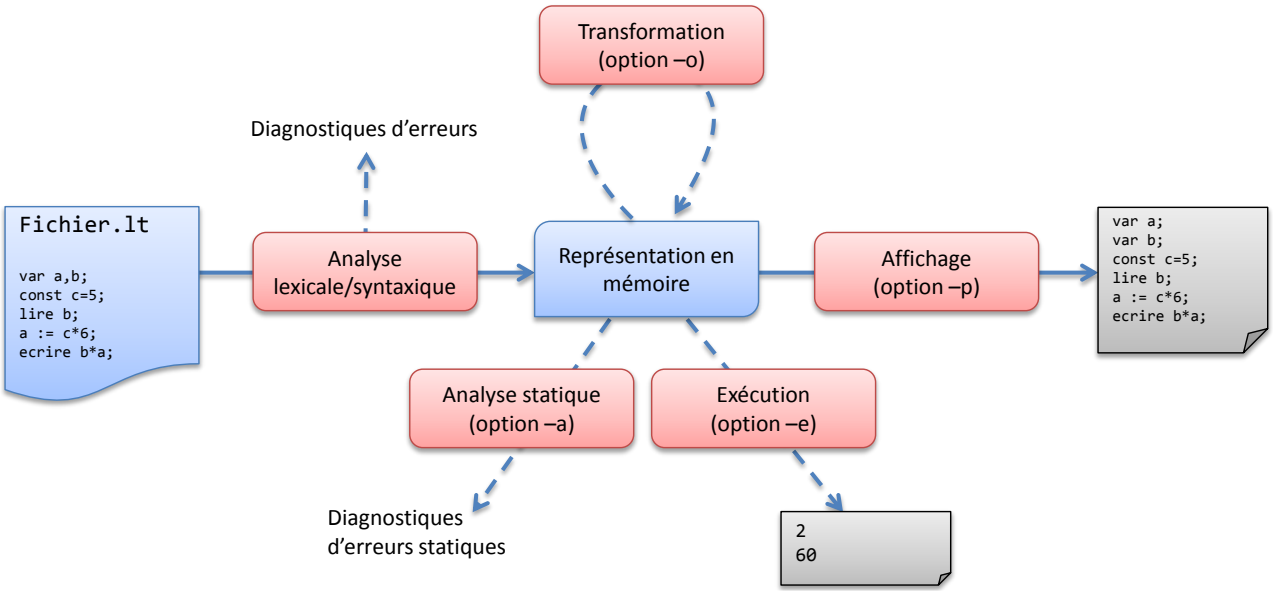
L'affectation est symbolisée par l'opérateur `:=`. En partie gauche doit être donné un identificateur de variable (la variable qui sera affectée) et en partie droite une expression.

Un identificateur est une suite de caractères alphanumériques commençant obligatoirement par une lettre. Les majuscules sont différenciées des minuscules. Un identificateur ne peut pas être l'un des mots-clés du langage : `var`, `const`, `ecrire` ou `lire`.

Une expression est une expression arithmétique composée d'identificateurs valides (variables ou constantes déclarées et affectées dans le cas d'une variable) et des opérateurs $\star$ (multiplication), + (addition), − (soustraction) et / (division). Tous les opérateurs sont associatifs à gauche. La multiplication ainsi que la division sont tous les deux prioritaires sur l'addition et la soustraction. Les parenthèses permettent de grouper des éléments afin de modifier localement ces priorités.

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). En complément, il existe quatre options en ligne de commande qui permettent d'enrichir ce fonctionnement de base :

1. Affichage du programme (option -p). Cette option affiche sur la sortie standard la représentation en mémoire du programme qui peut différer légèrement du code source mais dont la sémantique est respectée. Les éventuelles erreurs sont affichées sur la sortie d'erreur standard.
2. 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).
3. Exécution du programme (option -e). Cette option permet d'interpréter chacune des instructions du programme de manière interactive.
4. 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. Si l'option -p est activée, seul le programme transformé sera affiché.

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 faire vous-mêmes l'analyseur correpondant, donc qu'il ne faut pas multiplier les symboles non terminaux sans raison, et qu'il faut le plus possible être adapté à une analyse ascendante (privilégier les récursivités gauches par exemple). **Cette grammaire devra obligatoirement être validée par l'équipe enseignante afin que vous ne perdiez pas de temps à construire un automate pour rien.**

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 de la structure de données. Comment sera stocké votre programme ? Effectuez une conception en UML (diagramme de classe) qui permet de représenter en mémoire le programme. Il est impératif d'inclure dans ce diagramme de classes la partie liée à l'automate LR que vous aurez à implémenter. **Vous êtes invités à faire valider cette modélisation par l'équipe enseignante toujours dans le but de ne pas perdre de temps.**

Construction de l'automate LR. Cette tâche est fastidieuse et vous demandera de la concentration (ainsi qu'une feuille A3 !). Commencez par l'automate LR(0) si vous sentez que votre grammaire est LR(0). Si jamais vous avez des conflits, essayez de les résoudre plutôt d'une manière intuitive : à quoi correspond le choix issu du conflit ? Pensez aux priorités et associativité d'opérateurs.

Implémentation de l'analyseur lexical. À partir des expressions régulières identifiées précédemment, implémentez l'analyseur lexical en séparant bien les deux opérations de requête du prochain symbole et de déplacement de la tête de lecture.

Implémentation de l'automate LR. Implémentez l'automate LR grâce au *design pattern* **State** et aux structures de données imaginées précédemment. À chaque fois que cela sera possible, les vérifications sémantiques de base seront vérifiées au moment de la construction de la représentation en mémoire comme par exemple la détection de double déclaration.

Vérification statique du programme. Réfléchissez à l'algorithme qui va permettre de vérifier la validité du programme de manière statique. Voici les éléments qui sont demandés 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.
- Une variable n'a pas été déclarée.
- Une constante ne peut être modifiée.

Interprétation du programme. Cela correspond à l'exécution du programme mais à l'intérieur du programme d'analyse (donc sans construire d'éxécutable indépendant). L'instruction `lire` va engendrer une lecture au clavier, l'écriture va évaluer l'expression puis l'afficher, l'affectation va mettre à jour la table des symboles dynamique, *etc.*

Outil en ligne de commande - intégration. Il s'agit là de concevoir et implémenter l'outil qui va permettre d'interfacer l'analyseur. L'outil sera compatible avec le plan de test minimal proposé.

Transformation (facultatif mais recommandé). La représentation interne du programme peut être optimisée dans certains cas de figure. Les deux optimisations proposées ici sont :

1. Propagation de constantes. Une expression dont toutes les opérandes sont des constantes sera remplacée par la valeur résultante.
2. Éléments neutres. Chaque opérateur possède un élément neutre (l'addition le 0, la multiplication le 1, *etc.*), ces opérations peuvent être supprimées lorsqu'elles surviennent.

Gestion des erreurs lexicales (facultatif). Lorsqu'aucune expression régulière ne concorde avec l'entrée, cela signifie qu'il y a une erreur au niveau lexical. La façon la plus simple de gérer cette erreur et de supprimer le caractère suivant du flux et de chercher à nouveau.

Gestion des erreurs syntaxiques (facultatif). Si l'analyseur lexical ne fait pas d'erreur mais que le jeton renvoyé n'est pas conforme à la grammaire, alors l'automate est censé entrer dans un état d'erreur et ne peut plus continuer l'analyse. Une des manières de gérer cela est de modifier le comportement de la fonction de transition afin qu'elle puisse récupérer sur des erreurs simples. Par exemple, si un état nous dit qu'il faut forcément le symbole = à la suite et que ce symbole n'est pas présent alors on peut faire comme s'il avait été oublié et l'ajouter artificiellement. Beaucoup de petites erreurs peuvent être récupérées comme cela.

4 Plan de tests

Un plan de tests est fourni. Ce plan de tests est minimal et permet de vérifier que votre outil est bien conforme aux spécifications de base qui sont données (on peut même dire que le plan de test fait partie des spécifications). Vous avez la possibilité de le compléter, il se trouve sur moodle, dans le répertoire `Projet`, sous la forme d'une archive nommée `ProjetGLTests.tgz`.

Le framework de test suppose qu'un executable portant le nom `lut` se trouve dans le répertoire racine. S'il se trouve ailleurs, faites un lien symbolique.

5 Livrables (0 papier)

1. Un document de conception à déposer sur moodle qui contiendra
   — la grammaire du langage lutin que vous avez utilisée
   — l'automate LR en version graphique (pas forcément typographiée, vous pouvez scanner une version manuscrite)
   — la table des transitions de l'automate LR
   — la description des structures de données sous forme d'un diagramme de classes
2. Le 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 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 dans le formalisme du framework (et les fichiers nécessaires)
3. Une présentation orale montrant l'état de l'avancement du projet et l'exécution commentée des tests dans divers cas.

Tous ces livrables sont à rendre la semaine qui suit la dernière séance de projet.

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