TD 02 : Lexer & Parser

Ce second TD va nous permettre d'implémenter la première partie de notre compilateur : analyse lexicale et l'analyse syntaxique.

Les fichiers de base du compilateur vous sont fournis dans les ressources du cours.

1. Lexer

La quasi-totalité du code du lexer vous est déjà fourni dans les ressources du TD. Vérifier tout de même que rien ne manque et que tout vous semble bon.

2. Parser

C'est maintenant à vous de jouer et d'écrire les différentes règles d'analyse syntaxique afin que votre compilateur puisse lire votre fichier source. Nous vous proposons de découper cette étape de la manière suivante :

- Écrire les règles pour la déclaration de variable
- Écrire les règles pour l'accès aux variables (attention au tableau)
- Écrire les règles pour les opérations mathématiques
- Écrire les règles pour les expressions booléennes
- Écrire les règles pour les conditions, et boucle
- Écrire les règles pour l'appel de fonction
- Écrire les règles pour la déclaration de fonction

Si tout se passe bien, vous avez fini ;-)

3. Lexer – deuxième partie

Notre prochaine étape est d'écrire le code permettant de récupérer les différentes valeurs dont on a besoin en sortie du lexer. Par exemple, les identifiants de variables, les valeurs des entiers, ...

Dans le code appelé automatiquement par le Lexer lorsqu'il rencontre un des symboles que vous avez déclaré, vous disposez de trois variables :

- yyleng : contient la longueur de la chaine qui a été lu pour ce symbole
- yytext : contient la chaine qui a été lu. (Attention, les données seront détruites lorsque votre code aura fini de s'exécuter. Si vous souhaitez les garder, n'oubliez pas de les copier.)
- yylval : variable de type union qui vous permettra de stocker les informations que vous souhaitez garder. Ce type union est à déclarer dans votre fichier .y de la manière suivante :

%union { int intValue; char* strValue; } (vous pouvez bien entendu ajouter d'autre champs)

Ensuite, pour tous les tokens de votre lexer qui retourne un élément, vous devez définir au niveau du %token quel type est renvoyé. (Vous n'avez rien à ajouter pour les tokens qui ne renvoie rien)

Exemple avec les identifiants de variables :

%token <strValue> IDENTIFIER

4. Parser – Deuxième partie

Maintenant que vous avez pu récupérer les informations provenant du Lexer, votre but va être, à l'aide du parseur, de créer un AST (http://en.wikipedia.org/wiki/Abstract_syntax_tree) à partir de vos instructions.

Cet AST représente une forme abstraite du contenu du fichier source que vous avez lu. Par exemple, la déclaration d'une fonction va être un premier nœud dans l'arbre et les instructions contenues dans la fonction vont chacune représenter un nœud enfant de la fonction.

Dans un premier temps, nous vous proposons d'implémenter une classe Node permettant de gérer un nœud de l'arbre assez facilement. Cette classe devra permettre d'accéder aux enfants du nœud en question ainsi que d'avoir le type du nœud (IDENTIFIER, CONST_INT, IF, …). Vous devrez donc déclarer une enum pour ces types. À vous de voir ce qui est intéressant de garder. Vous penserez également à stocker la line et la colonne ou apparait le code relatif à ce nœud afin d'afficher des erreurs clairs par la suite si c'est nécessaire.

Après avoir implémenté cette classe Node, on se rend compte assez vite que nous allons avoir un problème pour stocker les valeurs (int, float, string…) donc certains de nos nœuds ont besoin. On vous propose donc de créer des classes enfants pour chaque type que nous aurions besoin de gérer. (IntNode, FloatNode, BoolNode, StrNode et tous ceux que vous jugerez intéressant)

Une fois ces classes définis, nous pouvons passez à l'implémentation du code qui sera exécuté après chaque règles du parseur et qui nous permettra de créer les différents nœuds de notre arbre.

Remarques

Dans l'archive qui vous est fourni avec le code de base, vous trouverez un makefile qui fonctionnera si vous développez en C++. Le fichier test.sh vous permet de compiler et tester votre compilateur simplement sur le fichier « sample » dans le répertoire « sample ». Afin que la compilation fonctionne, vous veillerez à bien mettre tous vos fichiers source dans le répertoire src.