PROJET D'ANALYSE SYNTAXIQUE
RAPPORT

SDAF | RAVENDRAN
2020 – 2021

TABLES DES MATIÈRES :

I. GUIDE D'UTILISATION

Pour compiler le projet et l'exécuter sur le fichier de votre choix, placez vous dans le répertoire source **src** , compiler le **makefile** avec la commande **make** puis **./as < ../tests/nom_du_fichier**.

Pour compiler le projet et l'exécuter sur les fichier TPC contenu dans les répertoires **valid** et **invalid**, exécutez le script **script.sh** avec la commande **./script.sh**.
Le script permet de comparer les **valeurs de retour** de l'analyseur sur le fichier et donc de nous informer si le **programme est correct** ou s'il y a une **erreur de compilation**.

L'analyseur utilise les outils **Flex** pour l'analyse lexicale et **Bison** pour l'analyse syntaxique.
L'analyseur lexical permet de renvoyer à Bison le token reconnu. Il renvoie le token de la chaîne la plus longue trouvé parmi toutes les règles. Il **renvoie 0 si un caractère interdit** est saisi.
L'analyseur syntaxique permet, lui, de parser les tokens et identifier dans la grammaire la règle à utiliser, ainsi que de les définir en indiquant la priorité. Il **renvoie 0** si le **programme analysé est correct** et **1 sinon**.

II. MÉTHODES UTILISÉES

1. Fonction affichage de l'erreur

Lorsqu'une erreur est détecté, la fonction **yyerror(char *s)** est appelé. Nous l'avons redéfini afin qu'elle affiche le message en couleur que l'on souhaite : numéro de ligne, type d'erreur et ligne de l'erreur avec une flèche verticale qui pointe sur l'erreur.
Le numéro de la ligne est compté à l'aide de l'analyseur lexical grâce à la variable **pos_line.** Celle-ci est incrémentée de 1 à chaque saut de lignes.
Pour l'affichage de la flèche au niveau de l'erreur, nous avons créé un compteur de caractère, **pos_char**, qui est incrémenté en fonction de la taille de la chaîne reconnu dans le fichier Flex, sinon le compteur est remis à zéro à chaque saut de ligne.

2. Récupération de la ligne d'erreur

Dans le fichier Flex, nous avons rajouté une règle qui permet de récupérer toute la ligne, ce qui sauvegarde la ligne dans la variable **yytext**. Afin de conserver la valeur dans le fichier bison, nous avons effectué une copie dans la chaîne de caractère **line,** qui est un champs de l'union créé dans l'analyseur syntaxique.

Une fois la copie effectué, nous avons fait appel a **REJECT**, qui permet de stopper l'analyseur et d'appeler la fonction **yyerror(char *s)**.

3. Valeur de retour de l'analyseur

L'analyseur **renvoie 0** si **aucune erreur** est détectée et **1 sinon**. Afin de savoir si une erreur à été détecté, on regarde la valeur de retour de **yyparse** et du nombre d'erreur trouvée. Si les deux valent 0, la valeur de retour est 0, sinon on renvoie 1. Avant de renvoyé la valeur, nous avons placé un message indiquant le nombre d'erreur.

III. MODIFICATION APRÈS PREMIER RENDU

1. Règle pour les littéraux de caractères

Nous avions mal géré les caractères, nous avons donc modifié la règle dans l'analyseur lexical afin que tout les lettres entre a-z, A-Z et tout les chiffres entre guillemets simple soit reconnu.
Nous avons utilisé la règle : **\'([a-zA-Z0-9]|\\n|\\'|\\t)\'**

2. Affichage de la valeur de retour

Au premier rendu, la valeur de retour n'était pas donnée (que le fichier soit compté comme correct ou non). Nous avons rajouté dans notre script la commande ci dessous pour récupérer et afficher la valeur attendue.

```
let "res = $?"
echo -e "Return value = \033[0;33m$res\033[m"
```

3. Répartition des fichiers

Dû à une mauvaise compréhension, nous avions fait deux dossiers tests, chacun comprenant deux fichiers tests à la limite de l'identique : un sans erreur et un autre avec. Aujourd'hui nous avons réorganisé nos dossiers : dans le dossier **tests** nous avons deux sous-dossiers intitulés **valid** et **invalid**. Chaque dossier comporte des fichiers TPC respectifs à leur nom de dossier (valid comporte des fichiers corrects et inversement).

4. Correction du conflit de décalage/réduction

En se basant sur les tokens, nous avons corrigé les problèmes présents dans la grammaire : l'analyseur syntaxique indique un conflit de décalage/réduction avant de parser **ELSE** car deux règles commencent de la même façon, il ne sait donc pas laquelle il doit appliquer .
%prec associe le token « **then »** au règle **IF'('Exp')'Instr**.

5. Autorisation des types de structures

Pour accepter les structures, nous nous sommes inspiré du fichier Bison. En regardant de plus près, on s'aperçoit que la grammaire des déclarateurs de fonction et de variable est utile à la création de la grammaire des structures. On assimile un token **STRUCT** à une grammaire nommée **DeclStruct** : elle est définie par un token **STRUCT** et un **IDENT** suivi de son corps **CorpsStruct**. Le corps est composé d'accolade ouvrante et fermante et contiennent les déclarations des variables **DeclVars**. Si la structure repérée ne contient pas de corps, on regarde l'autre grammaire possible : **STRUCT IDENT Declarateurs ';'**, elle permet de lire une structure appelé par un ou plusieurs **Déclarateurs** qui sont lié au token **IDENT**.

```
Prog: Decls DeclFoncts ;

Decls: Decls TYPE Declarateurs ';'
     | Decls DeclStruct
     | ;

DeclStruct: STRUCT IDENT CorpsStruct
     | STRUCT IDENT Declarateurs ';' ;

CorpsStruct: '{' DeclVars '}' ';' ;
```

Pour pouvoir appeler plusieurs règles à la suite, on a fait une grammaire **Decls** qui à le choix entre appeler un token TYPE suivi d'un Déclarateur et d'un ' ;' qui indique l'appel d'une variable (ex : int a;) ou rappeler la grammaire **DeclStruct**. Pour finir dans le fichier Bison, on a modifié la grammaire de **Prog** : de cette manière, on peut avoir un fichier TPC qui commence par soit des variable globale, soit des structure à la suite plusieurs fois et se termine par les déclarations de fonctions. Puis, dans notre analyseur lexical, nous avons rajouté une règle qui nous permet de reconnaître une structure :

%struct {pos_char += yyleng; return STRUCT;}.

6. Modification du script

On a modifié notre script, de sorte que, les **noms des fichiers** ainsi que leurs **résultats** (correct ou incorrect) de l'analyseur soient sauvegardé dans un fichier **result.txt**. On y a aussi ajouté le nombre de résultats correct et incorrect en fin de fichier.

IV. AMÉLIORATIONS

1. Reprise à l'erreur

Nous avons utilisé **yyclearin** pour récupérer une erreur. Nous avons donc, ajouté à la grammaire une règle **error** qui appelle **yyclearin : | error { yyclearin;} ;**.
Lorsqu'une erreur est trouvée, l'analyseur repartira de la où il s'est arrêté.

Cependant, la règle **yyclearin** assigne à la valeur de retour de **yyparse** de toujours valoir 0. C'est pourquoi, nous avons décidé de compter le nombre d'erreur rencontré lors de l'analyse avec un **compteur** qui sera incrémenté de 1 à chaque appel de **yyerror**.

2. Affichage du résultat

Afin de permettre une plus grande visibilité des résultats de l'analyseur, nous avons écrit en rouge le type de l' erreur, suivi de la ligne et colonne près de l'erreur. La valeur de retour est affichée en jaune et une phrase clignotante en rouge/vert qui indique si le programme est incorrect/correct.

3. Affichage du nombre d'erreurs

Comme le nombre d'erreur dans un fichier a été traité, nous avons décider d'afficher le nombre de ceux-là dans le programme, à la fin de la lecture du fichier TPC.

V. DIFFICULTÉS RENCONTRÉS

L'un des problème rencontré est celui de la mise en place de la reprise à l'erreur. Nous avons essayé plusieurs méthodes : en utilisant uniquement yyerrork, puis en utilisant yyerrork et yycleanin. La recherche de documentation concernant ce problème a été compliquée. Nous avons donc opté pour une méthode sans qu'elle ne soit totalement optimale.
L'autre problème rencontré est celui concernant la permission d'avoir dans un fichier TPC plusieurs règles se mélangeant : une variable globale suivi par une structure suivi d'une autre variable globale suivi d'une structure globale etc. finissant par les déclarations de fonctions. Nous arrivions seulement à avoir soit des structure soit des variables mais jamais les deux ensemble en début de fichier. Au départ on avait cette grammaire concernant la structure :

```
DeclStruct: STRUCT IDENT CorpsStruct
        | STRUCT IDENT Declarateurs ';' ;

CorpsStruct: '{' DeclVars '}' ';' ;

DeclStructs: DeclStructs DeclStruct
                | DeclVars ;

Prog: DeclStructs DeclVars DeclFoncts
        | DeclVars DeclFoncts ;
```

notre fichier qui contenait au début des variables globales et ensuite que des structures jusqu'à arrivé aux déclarations de fonctions, était correct pour l'analyseur syntaxique , mais, dès lors qu'on avait une variable globale entre deux structures, celui-ci retournait une erreur. Ce problème a finalement été résolu.