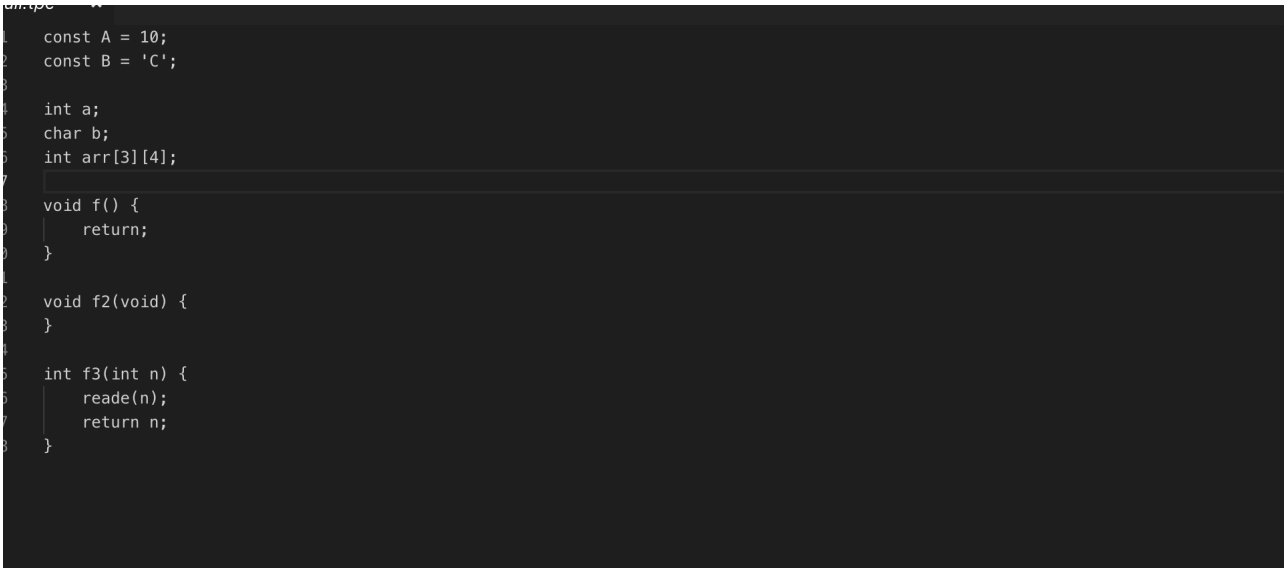
Rapport de projet

Compilation 2018-2019

Mamadou Cisse - Volkan Zulal

Présentation

L'objectif de ce projet est la réalisation d'un compilateur à l'aide des outils flex et bison pour le langage « TPC » inspiré de la syntaxe du C.



```
const A = 10;
const B = 'C';

int a;
char b;
int arr[3][4];

void f() {
    return;
}

void f2(void) {
}

int f3(int n) {
    reade(n);
    return n;
}
```

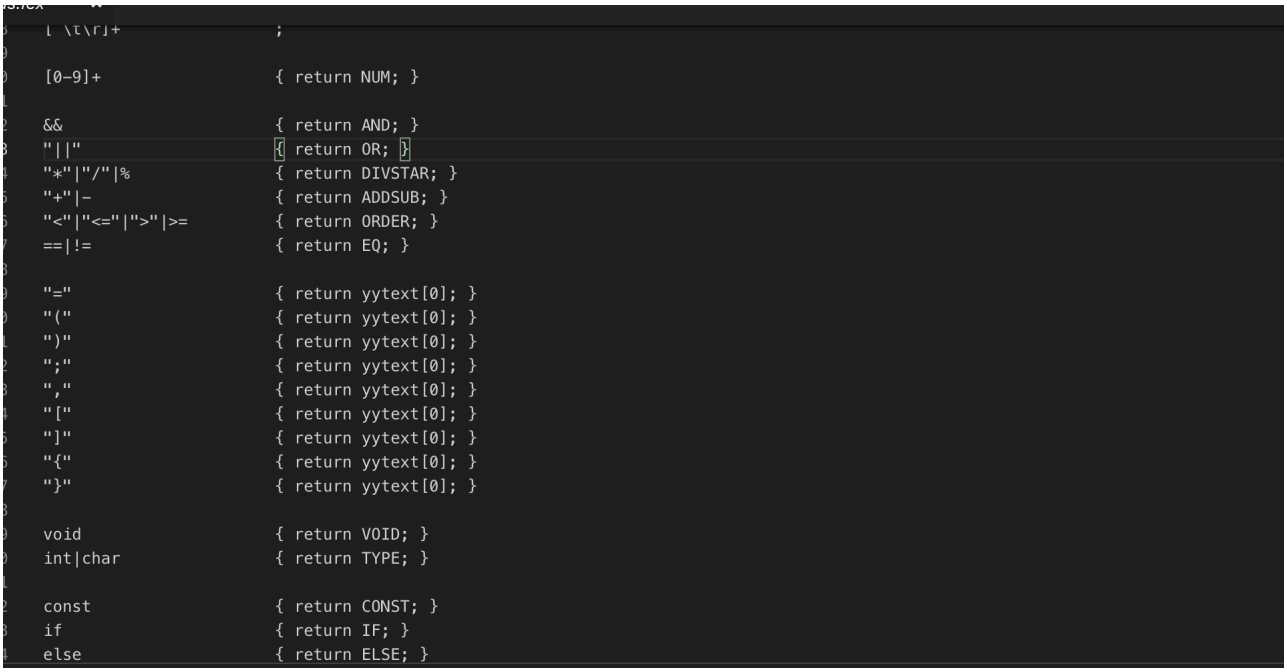
Exemple d'un programme écrit dans le langage TPC

Fonctionnement

Le projet est séparé en 3 parties :

- Une partie d'analyse lexicale réalisé avec flex
- Une partie d'analyse syntaxique avec bison
- Une partie de tests unitaires en bash

Analyse lexicale



```
[ \t\r]+                ;

[0-9]+                  { return NUM; }

&&                      { return AND; }
"||"                    { return OR; }
"*"|"/"|%               { return DIVSTAR; }
"+"|-                   { return ADDSUB; }
"<"|"<="|">"|>=         { return ORDER; }
==|!=                   { return EQ; }

"="                     { return yytext[0]; }
"("                     { return yytext[0]; }
")"                     { return yytext[0]; }
";"                     { return yytext[0]; }
","                     { return yytext[0]; }
"["                     { return yytext[0]; }
"]"                     { return yytext[0]; }
"{"                     { return yytext[0]; }
"}"                     { return yytext[0]; }

void                    { return VOID; }
int|char                { return TYPE; }

const                   { return CONST; }
if                      { return IF; }
else                    { return ELSE; }
```

Notre fichier d'analyse lexicale as.lex

La partie d'analyse lexicale du projet a été réalisée à l'aide de l'outil Flex en langage C. On a crée un fichier as.lex. Dans ce fichier on a déclaré nos règles flex nous permettant de reconnaître les différents tokens constituant le langage TPC. Toutes les règles sont reconnues à l'aide d'expression régulières sauf la règle permettant de reconnaitre un commentaire qui elle utilise les conditions de démarrages. Ce choix s'explique par le fait que notre analyseur affiche les lignes contenants des erreurs de syntaxes, pour cela on a besoin d'une règle permettant de détecter le caractère '\n'. Comme les commentaires du langage TPC peuvent être sur plusieurs lignes, Cela aurait était compliqué de garder une référence sur la ligne courante de l'analyseur avec des expressions régulières.

Le fichier contient aussi une règle '.' à la toute fin pour détecter et afficher les caractères inconnus. Notre programme flex ne reconnait que les tokens de bases du langage, on a ajouté aucun autre token.

Analyse syntaxique

```
int lineno;
int err_count = 0;
%}

%token COMMENT
%token NUM
%token AND
%token OR
%token DIVSTAR
%token ADDSUB
%token ORDER
%token EQ
%token TYPE
%token CONST
%token IF
%token ELSE
%token WHILE
%token PRINT
%token READC
%token READE
%token RETURN
%token IDENT
%token VOID
%token CARACTERE

%%
Prog:  DeclConsts DeclVars DeclFoncts
```

Notre fichier d'analyse syntaxique as.y

Notre projet utilise l'outil bison comme analyseur syntaxique. Le code source de l'analyseur est placé dans le fichier as.y.

Dans ce programme bison dont la plupart des règles de grammaire nous ont été donné dans le sujet du projet est celle qui contient la partie du sujet qui nous posait un peu de problème. En effet au début du projet on n'avait pas trop bien compris la syntaxe de bison donc on est restés bloqués un petit moment dessus. Mais une fois l'outil maitrisé, on a comprit qu'il fallait déclarer les tokens comme sur la capture d'écran ci-dessous pour que le programme bison puisse les associés aux règles du programme flex.
Comme pour la partie analyse lexicale, on a ajouté que les tokens de bases du langage.
Mais contrairement au flex, notre programme bison contient quelques améliorations :

- reconnaissance des tableaux à plusieurs dimensions à l'aide de la règle « **Tableau** ».
- possibilité de continuer l'analyse syntaxique en cas d'erreur grâce à l'ajout du token prédéfinie « error » dans les règles pouvant contenir des erreurs de syntaxe.
- définition de la fonction « **yyerror()** » afin d'afficher les lignes contenants des erreurs de syntaxe sur la sortie d'erreur standard.

Generation Nasm

Pour générer le code nasm nous utilisons le module « **emit** » composé des fichiers emit et emit.c . Ce module contient des fonctions générant différentes instructions nasm en utilisant les données en paramètre ainsi que la table des symboles du module « **symbs** »

Tests unitaires

Le projet est aussi livré avec quelques tests unitaires dans le dossier **tests**.
Il est possible de lancer les tests en exécutant le script bash « test.sh » une fois que le projet est compilé avec la commande make.
Il y a trois dossiers de tests:

- Le dossier « **good/** » qui contient des programmes écrits en TPC ne contenants aucune erreur de syntaxe.
- Le dossier « **bad/** » qui contient des programmes écrits en TPC contenants quelques erreurs de syntaxe.
- Le dossier « **recovery/** » qui contient des programmes écrits en TPC contenants quelques erreurs de syntaxe qui ne terminent pas l'analyse syntaxique.

L'exécution de ces tests génère un fichier **log.txt** contenant le détail des tests.

Conclusion

Nous avons réalisé la totalité du projet en ajoutant quelques améliorations. Nous n'avons pas eu de problème en particulier mis à part sur les conflits de l'outil bison au début du projet ainsi l'implémentation des tableaux.