# PROJET ANALYSE SYNTAXIQUE

Decembre 2023

TOUATI

Damien

Sommaire:

## INTRODUCTION :

Le projet d'analyse syntaxique à pour but de d'apprendre à utiliser les outils flex et bison et savoir les combiner pour fabriquer un analyseur lexical pour un petit langage de programmation (le langage TPC).

## DIFFICULTÉS RENCONTRÉES :

Il a été difficile de construire l'arbre abstrait de cette grammaire en raison de sa grande taille.

Pour pouvoir commencer les premiers tests il faut avoir renvoyé le bon type pour chaque règle de la grammaire ce qui veut dire qu'il faut d'abord écrire toutes les instructions qui vont permettrent la construction de l'arbre abstrait et ensuite corriger les erreurs en effectuant de multiples tests. C'est là que l'on rencontre une autre des difficultés du projets c'est-à-dire qu'il faut trouver des tests à la fois pertinents et qui ont un but. Par exemple ce test ci :

```
int a;
int main(void){
    if(i == 1){
        a = 2;
    }
    else{
        a = 3;
    }
    return 1;
}
```

Il me permet d'analyser le structure de mon arbre pour la condition : if-else
Donc ce test me permet à la fois de bien construire mon arbre abstrait ainsi que répondre au besoin du sujet de fournir une batterie de tests.

## MANUEL UTILISATEUR:

Pour la compilation il suffit de taper la commande make dans le terminal en vous trouvant dans le répertoire du projet.
Pour ce qu'il s'agit de l'exécution du programme, partons du principe que vous vous trouvez dans le répertoire de base là où se trouve tous les sous-dossiers, il y a plusieurs possibilités:

-Vous voulez utiliser votre propre programme tpc pour tester l'analyseur syntaxique alors deux choix s'offrent à vous. Vous pouvez soit faire votre propre fichier tpc et utiliser la commande : "./bin/tpcas [OPTIONS] < votrefichier.tpc" ou bien utiliser la commande ./bin/tpcas et taper directement votre programme dans la ligne de commande.
Le premier choix vous permettra d'utiliser les deux options qui s'offrent à vous et qui sont purement optionnelles, l'option d'aide qui affiche toutes les commandes à savoir pour le bon fonctionnement du programme et pour l'utilisateur est l'option "-h" ou "--help", l'option d'affichage de l'arbre fera apparaître l'arbre abstrait de votre programme tpc dans le terminal avec la commande "-t" ou "--tree". Ces deux options sont utilisables simultanément.
Avec ce deuxième choix vous ne pourrez pas utiliser d'option car pour sortir du programme il faudra que vous fassiez une erreur de syntaxe tant que ça ne sera pas le cas il vous faudra continuer votre code ou sortir de force avec la commande CTRL+C.
Une batterie de tests est aussi fournie dans le projet et encore ici deux choix s'offrent à vous. Vous pouvez utiliser un des multiples tests mis à votre disposition pour tester l'analyseur syntaxique ou bien vous pour utiliser le fichier tests.sh qui testera pour vous tous les fichiers .tpc et affichera le score de tous les tests dans le terminal.
Bien sûr vous pourrez retrouvez ces informations en utilisant l'option -h ou –help.

## Solutions techniques proposées et justifications des choix effectués:

La grammaire fournie dans le projet comporte une ambiguïté:

```
projet.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
projet.y: warning: shift/reduce conflict on token ELSE [-Wcounterexamples]
  Example: IF '(' Exp ')' IF '(' Exp ')' Instr • ELSE Instr
  Shift derivation
    Instr
    ↳ 23: IF '(' Exp ')' Instr
                         ↳ 24: IF '(' Exp ')' Instr • ELSE Instr
  Reduce derivation
    Instr
    ↳ 24: IF '(' Exp ')' Instr                       ELSE Instr
                         ↳ 23: IF '(' Exp ')' Instr •
```

Cette ambiguïté vient du fait qu'il peut y avoir deux avoir arbres de dérivations pour une même instructions, ici :

IF '(' Exp ')' IF '(' Exp ')' ELSE Instr

On peut la traduire par le fait qu'on ne sait pas à quel IF raccordé le ELSE.

La convention établie est que ces ambiguïtés sont résolues en attachant la clause else à l'énoncé if le plus interne et c'est ce que Bison accomplit en choisissant de "shift" plutôt que de "reduce". (Il serait idéalement plus propre d'écrire une grammaire non ambiguë, mais c'est très difficile à faire dans ce cas)

La solution pour pallier ce problème est de dire à bison que ce genre de conflit prévisible et légitime n'a pas besoin d'être traité et qu'il faut utiliser la déclaration "%expect n" qui cachera les n avertissements tant que le nombre de conflits de shift/reduce est exactement n.

```
%token <ident> TYPE VOID IF ELSE WHILE RETURN OR AND CHARACTER ADDSUB DIVSTAR IDENT ORDER EQ
%token <num> NUM

%type <node> DeclVars Prog Declarateurs DeclFonct DeclFoncts EnTeteFonct Parametres ListTypVar Corps SuiteInstr Instr
%type <node> TB FB Exp M E T F LValue Arguments ListExp

%expect 1
```

De plus je voudrais vous faire part de la justifications des choix effectués comme par exemple ma structure Node et l'union de mon tpcas.y:

```
typedef struct Node {
  label_t label;
  struct Node *firstChild, *nextSibling;
  int lineno;
  int num;
  char * ident;
} Node;
```

```
%union{
    Node* node;
    int num;
    char ident[64];
}
```

Ce choix permet de rester sur une structure assez simple sans trop d' attributs avec pour les non terminaux le type node et les terminaux de type ident sauf pour le non terminal NUM qui sera de type num.

C'est aussi plus simple de récupérer les attributs depuis le fichier tpcas.lex car il suffit de faire un strcpy pour tous les attributs que l'on veut récupérer à part pour num du coup où il faut faire un yylval = atoi(yytext);

```
"int" {ch+= yyleng; strcpy(yylval.ident, yytext); return TYPE;}
"char" {ch+= yyleng; strcpy(yylval.ident, yytext); return TYPE;}

[0-9]+ {ch+= yyleng; yylval.num = atoi(yytext); return NUM;}

'[^']' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}
'\\n' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}
'\\t' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}
'\'' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}
```

Pour ce qui est du fichier tpcas.lex j'ai volontairement fait plusieurs règle simple qui renvoie le même token au lieu d'un règle complexe qui renvoie le token comme ci-dessous:

```
'[^']' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}
'\\n' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}
'\\t' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}
'\'' {ch+= yyleng; strcpy(yylval.ident, yytext); return CHARACTER;}

[A-Za-z_][A-Za-z0-9_]* {ch+= yyleng; strcpy(yylval.ident, yytext); return IDENT; }

"==" {ch+= yyleng; strcpy(yylval.ident, yytext); return EQ;}
"!=" {ch+= yyleng; strcpy(yylval.ident, yytext); return EQ;}

"<" {ch+= yyleng; strcpy(yylval.ident, yytext); return ORDER;}
"<=" {ch+= yyleng; strcpy(yylval.ident, yytext); return ORDER;}
">" {ch+= yyleng; strcpy(yylval.ident, yytext); return ORDER;}
">=" {ch+= yyleng; strcpy(yylval.ident, yytext); return ORDER;}

"+" {ch+= yyleng; strcpy(yylval.ident, yytext); return ADDSUB;}
"-" {ch+= yyleng; strcpy(yylval.ident, yytext); return ADDSUB;}

"/" {ch+= yyleng; strcpy(yylval.ident, yytext); return DIVSTAR;}
"*" {ch+= yyleng; strcpy(yylval.ident, yytext); return DIVSTAR;}
"%" {ch+= yyleng; strcpy(yylval.ident, yytext); return DIVSTAR;}
```

J'ai aussi rajouté des noeuds explicite dans l'arbre abstrait pour aider à la compréhension de l'arbre pour l'utilisateur:

```
Prog:  DeclVars DeclFoncts {
    node = makeNode(Prog);
    Node * declV = makeNode(Global_vars);
    Node * declFuncs = makeNode(Functions);
    addChild(node, declV);
    addChild(declV, $1);
    addChild(node, declFuncs);
    addChild(declFuncs, $2);
    }
    ;
```

```
Prog
├── global_vars
│   ├── int
│   │   ├── a
│   │   ├── b
│   │   ├── c
│   │   ├── d
│   │   ├── e
│   │   ├── f
│   │   ├── g
│   │   ├── h
│   │   ├── i
│   │   ├── j
│   │   ├── k
│   │   ├── l
│   │   └── m
│   ├── int
│   │   └── a
│   └── int
│       └── b
└── Functions
    └── Function
        ├── int
        ├── main
        ├── Parameters
        │   └── Void
        └── Corps
            └── instructions
                └── return
                    └── 0
```

```
Prog
├── global_vars
└── Functions
    └── Function
        ├── int
        ├── main
        ├── Parameters
        │   ├── int
        │   │   └── argc
        │   └── array
        │       └── char
        │           └── argv
        └── Corps
            └── instructions
                └── while
                    ├── 1
                    └── instructions
                        └── return
                            └── 1
```

Comme ci-dessus on peut voir que je rajoute deux nœuds global_vars et functions qui permet de faire la différence entre la variables globales et la déclarations des fonctions plus bas, par exemple dans la deuxième image on voit que les déclarations des variables globales et on comprend vite que ce sont des variables globales.

Et en regardant la troisième image comprend vite qu'il n'y a pas de variables globales dans le programme

## CONCLUSION:

En conclusion, ce projet a été très enrichissant et le sera encore plus à l'avenir car il me permettra de construire mon premier compilateur pour le langage tpc, grâce à cette base solide qui est ce projet.