Rapport d’ANALYSE syntaxique.

2024-2025

Tatiana Desravines

Sommaire.

Rapport d'analyse syntaxique

01 Introduction.

Dans le cadre de ma formation en Licence 3 d'Informatique, j'ai réalisé une analyse syntaxique portant sur un compilateur de sous-programmes en langage C, appelé TPC.

Ce travail avait pour objectif de comprendre et de mettre en pratique les principes fondamentaux de l'analyse syntaxique, un élément clé dans le domaine des compilateurs et des langages de programmation.

Ce rapport présente les différentes étapes suivies pour réaliser cette analyse, les difficultés rencontrées, ainsi que les solutions apportées. Pour en savoir davantage sur le mode d'utilisation du compilateur, veuillez consulter le fichier *README*.

02 Contexte et objectifs.

2.1. Définition de l'analyse syntaxique

L'analyse syntaxique, ou parsing, consiste à vérifier que la structure d'un programme source respecte les règles de grammaire d'un langage donné.

Elle s'inscrit dans le processus de compilation ou d'interprétation d'un programme, après l'analyse lexicale, et avant la génération de code.

2.2. Objectifs du projet

Ce projet vise à :

- Implémenter un analyseur syntaxique capable de détecter les sous-programmes en langage C qui respectent les règles de syntaxe.

- Approfondir la compréhension des approches d'analyse syntaxique, notamment l'analyse ascendante.

- Utiliser des outils de génération d'analyseurs, comme Flex et Bison, pour automatiser le processus d'analyse syntaxique.

2.3. Analyse lexicale

L'analyse lexicale constitue une étape préalable à l'analyse syntaxique. Elle permet de transformer le texte source en une suite de tokens (unités lexicales) en se basant sur des règles définies.

Dans le cadre de ce projet, Flex a été utilisé pour générer l'analyseur lexical, facilitant ainsi l'extraction des éléments nécessaires à l'analyse syntaxique.

03 Méthodologie.

3.1. Choix de la grammaire

La grammaire utilisée dans ce projet nous a été imposée. Elle définit les règles du langage prises en charge par Bison.

Cependant, certaines règles ont été modifiées ou ajoutées pour répondre aux exigences spécifiques du projet.

Règle modifiée

```
Corps: '{' DeclVarLocalList  SuiteInstr '}' {

                                $$ = makeNode(bodyFunct , type);
                                addChild($$ , $2);
                                addSibling($2 , $3);

                               }
```

La grammaire ci-contre a été modifié pour pouvoir manipuler les variables locales.

Nous avons simplement remplacé *DeclVars* par *DeclVarLocalList*.

Image 1 :
Modification de la grammaire

Règle ajoutée

```
180
181  DeclVarLocal:
182      DeclVarLocal DeclVarL ';'  {
183         addChild($$ , $2);
184      }
185      | DeclVarL ';' {
186          $$ = $1;
187      }
188
189      ;
190
191
192
193
194
195
196
197
```

Image 2 :
Implémentation d'une action récursive

```
DeclVarL:
    STATIC TYPE Declarateurs {
        Type varType;
        strcpy(varType.declStatic, $1);
        strcpy(varType.typeVar, $2);
        Node * typeDecl = makeNode(typeVariable , varType);
        Node * typeVisibility = makeNode(visibilityStatic , varType);
        $$ = $3;
        addChild($$ , typeDecl);
        addSibling(typeDecl , typeVisibility);


    }       %type <node> Declarateurs
    | TYPE Declarateurs {
        Type varType;
        strcpy(varType.typeVar, $1);
        Node * typeDecl = makeNode(typeVariable , varType);
        $$ = $2;
        addChild($2, typeDecl);


    }
    ;
```

Image 3 :
Ajout de la visibilité local appelé static

```
DeclVarLocalList :
    DeclVarLocal {
                    $$ = makeNode(declarationListLocal , type);
                    addChild($$ ,$1);
                }
    | {$$ = makeNode(declarationListLocal , type);
      }
    ;
```

Image 4 :
Implémentation de la liste de variable local

La grammaire ci-contre permet de d'implémenter toutes les variables locales d'une fonction.

On gère également le cas où la fonction ne contient pas de variable locale.

3.2. Règle lexicale définie

L'analyse lexicale permet de transformer le code source en une séquence de tokens qui seront utilisés par l'analyse syntaxique.

Les catégories lexicales définies dans ce projet incluent les identifiants, les mots-clés, les opérateurs, les constantes.

```
%%
"/*"              {BEGIN(COMMENTAIRE);}
"//".*       ;
[\t ]        ;
\n  {lineno++; nbCarac = 0;}
[*/%]          {nbCarac++;yylval.operation = yytext[0];return(DIVSTAR);}
[0-9]+      {nbCarac++;yylval.num = atoi(yytext);return(NUM);}
"static"         {nbCarac++;strcpy(yylval.declStatic , yytext) ;  return(STATIC);}
"void"           {nbCarac++;strcpy(yylval.typeVar , yytext); return(VOID);}
("int"|"char")  {nbCarac++;strcpy(yylval.typeVar , yytext) ;return(TYPE);}
"while"          {nbCarac++;return(WHILE);}
"return"         {nbCarac++;return(RETURN);}
"else"           {nbCarac++;return(ELSE);}
"if"           {nbCarac++;return(IF);}
[A-Za-z_][a-zA-Z0-9_]* {nbCarac++;strcpy(yylval.ident , yytext); return(IDENT);}
\'\\(n|t|0)\'    {nbCarac++;strcpy(yylval.caracter , yytext) ;return(CHARACTERE);}
\'.\'       {nbCarac++;strcpy(yylval.caracter , yytext);return(CHARACTERE);}
(==|!=)           {nbCarac++;strcpy(yylval.ordre , yytext);return(EQ);}
(<|>|<=|>=) {nbCarac++;strcpy(yylval.ordre , yytext);return(ORDER);}
[-+]        {nbCarac++;yylval.operation = yytext[0]; return(ADDSUB);}
&&            {nbCarac++;strcpy(yylval.ordre , yytext);return(AND);}
\|\|           {nbCarac++;strcpy(yylval.ordre , yytext);return(OR);}
.             {nbCarac++;return yytext[0];}
```

Voici un extrait des règles lexicales définies ci-dessus,

exprimées à l'aide d'expressions régulières :

04 Difficultés rencontrées et solutions.

Problème

Lors de la gestion des structures conditionnelles (if-else), un conflit de type "empiler/réduire" a été détecté.

Ce type de conflit est courant dans les analyseurs syntaxiques générés par **Bison**, lorsque plusieurs règles grammaticales peuvent s'appliquer à une même séquence de tokens.

En l'occurrence, le problème s'est posé lors de l'analyse de structures conditionnelles imbriquées, où deux règles grammaticales entraient en conflit. En utilisant une table **LR(0)**, j'ai constaté qu'un conflit apparaissait entre les branches conditionnelles imbriquées du type if (condition) { ... } else { ... }, où l'analyseur ne savait pas s'il devait réduire ou empiler les règles en fonction des tokens rencontrés.

L'avertissement généré par **Bison** a signalé la règle concernée, précisant qu'une ambiguïté dans la grammaire entraînait une indétermination sur l'action à réaliser.

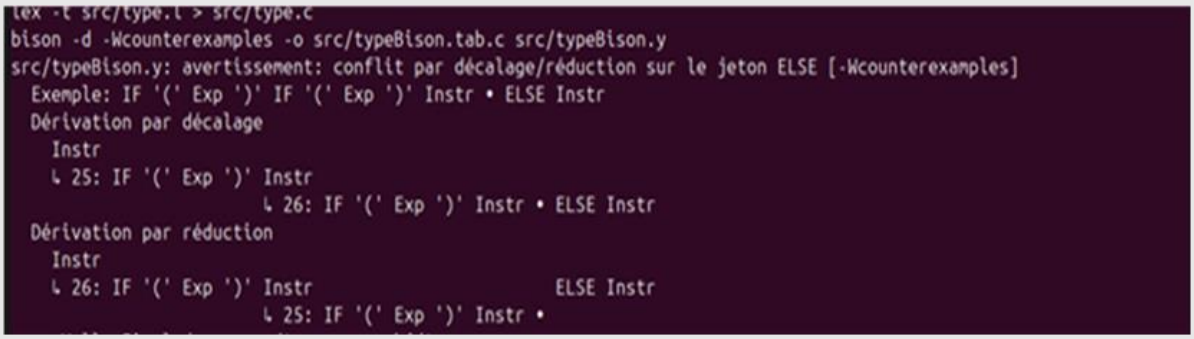

Image 5 : Exemple de conflit généré

```
261     |   IF '(' Exp ')' Instr {$$ = makeNode(branch , type);
262                             addChild($$ , $3);
263                             addSibling($3 , $5);
264
265                            }
266
267
268     |   IF '(' Exp ')' Instr ELSE Instr{$$ = makeNode(branch , type);
269                                         addChild($$ , $3);
270                                         Node * brancheIf = makeNode(ifBody , type);
271                                         addChild(brancheIf , $5);
272                                         Node * brancheElse = makeNode(elseBody, type);
273                                         addChild(brancheElse , $7);
274                                         addSibling($3 , brancheIf);
275                                         addSibling($3 , brancheElse);
276                                        }
277
278
```

Image 6 : la grammaire ambiguë

Solution

Le programme Bison traite par défaut les règles dans l'ordre dans lequel elles sont définies. Cela garantit que la bonne règle est prioritaire. Pour résoudre ce problème, j'ai utilisé la directive %expect pour indiquer au compilateur de ne pas signaler ce conflit comme une erreur. Cela supprime les avertissements inutiles tout en maintenant le fonctionnement correct du programme.

Problème

Lors de la gestion des déclarations de variables locales dans les fonctions, il a été nécessaire de traiter spécifiquement les variables static.

Le principal défi consistait à créer une règle grammaticale capable de reconnaître et d'accepter les variables static dans le corps des fonctions, tout en permettant une gestion récursive correcte des déclarations.

Cela a entraîné un conflit entre les règles lorsque l'on tentait de rendre la grammaire récursive pour accepter à la fois les variables static et non static sans introduire d’ambiguïtés.

En particulier, la difficulté est survenue lorsqu'il fallait accepter à la fois les variables locales ordinaires et celles déclarées comme static dans une même fonction, tout en maintenant la récursivité dans les déclarations.

Solution

Pour résoudre ce problème, j'ai introduit trois symboles distincts dans la grammaire :

1. DeclaVarL : Ce symbole regroupe tous les types de variables, qu'elles soient static ou non. Il permet de gérer de manière uniforme les variables avec ou sans le modificateur static, tout en assurant la cohérence de la grammaire

   (Pour plus de détail sur l'implémentation voire image 3 : Ajout de la visibilité local appelé static section 3. Méthodologie).

2. DeclaVarLocal : Un autre symbole a été introduit pour gérer la récursion des déclarations dans le corps des fonctions. Cela permet de définir des règles qui gèrent spécifiquement les déclarations de variables locales à l'intérieur des fonctions sans entrer en conflit avec celles des variables static

   (Pour plus de détail sur l'implémentation Image 2 : Implémentation d'une action récursive).

3. DeclaVarLocalList : Ce symbole est utilisé pour déclarer la liste des variables locales d'une fonction, et il prend en compte le fait qu'une fonction peut ne pas avoir de variables déclarées, offrant ainsi plus de flexibilité dans les cas où aucune déclaration n'est présente.

   (Pour plus de détail sur l'implémentation Image 4 :

   Implémentation de la liste de variable local).

05 Enseignements tirés et conclusion.

Enseignements tirés

Ce projet m’a permis de :
Renforcer ma compréhension des concepts fondamentaux de l’analyse syntaxique.
Apprendre à utiliser des outils professionnels comme Bison.
Améliorer mes compétences en résolution de problèmes et en conception de grammaires.

Cela supprime les avertissements inutiles tout en maintenant le fonctionnement correct du programme.

Conclusion

L’analyse syntaxique est une étape incontournable dans la conception des compilateurs et des interpréteurs.

Ce projet m’a fourni une base solide pour poursuivre mes études en théorie des langages et compilation.

Je remercie enseignants pour leur accompagnement.

Annexes

ARBRE SYNTAXIQUE EXEMPLE :

```
Program
├─ declaration-list
└─ Functions
    └─ Function
        ├─ Heading
        │   ├─ return.type: int
        │   ├─ id.name = main
        │   └─ Parameters
        │       └─ id.type: void
        └─ Body
            ├─ InnerDeclarations
            │   └─ id.name = c
            │       └─ id.type: int
            └─ InstructionBlock
                ├─ Assign
                │   ├─ id.name = c
                │   └─ Binary  (op: +)
                │       ├─ Binary  (op: -)
                │       │   └─ value: 4
                │       └─ Binary  (op: *)
                │           ├─ id.name = c
                │           └─ value: 4
                └─ return
                    └─ value: 0
```

CODE SOURCE EN C DE L'IMPLEMENTATION DES OPTIONS :

```c
void isOptionShort(int argc , char * argv[], int * optH ,int * optT){
    int opt = 0 ;



    for(;(opt = getopt(argc , argv, ":h::t")) != -1; ){

        switch(opt){
            case 't':
                *optT = 1;
                break;
            case 'h':
                *optH = 1;
                break;
            case '?':
                return;

        }


    }


}
```

```c
void isOptionLong(int argc , char * argv[] , int *optH , int *optT){
    if(*optH && *optT)
        return;


    for (int i = 0; i < argc; i++)
    {

        if(!strcmp(argv[i], "--tree") && !(*optT)){
            *optT = 1;

        }
        else if(!strcmp(argv[i], "--help") && !(*optH)){
            *optH = 1;

        }


    }


}
```