# Documentation développeur

## H4222

# Fonctionnalités

## Grammaire et analyse

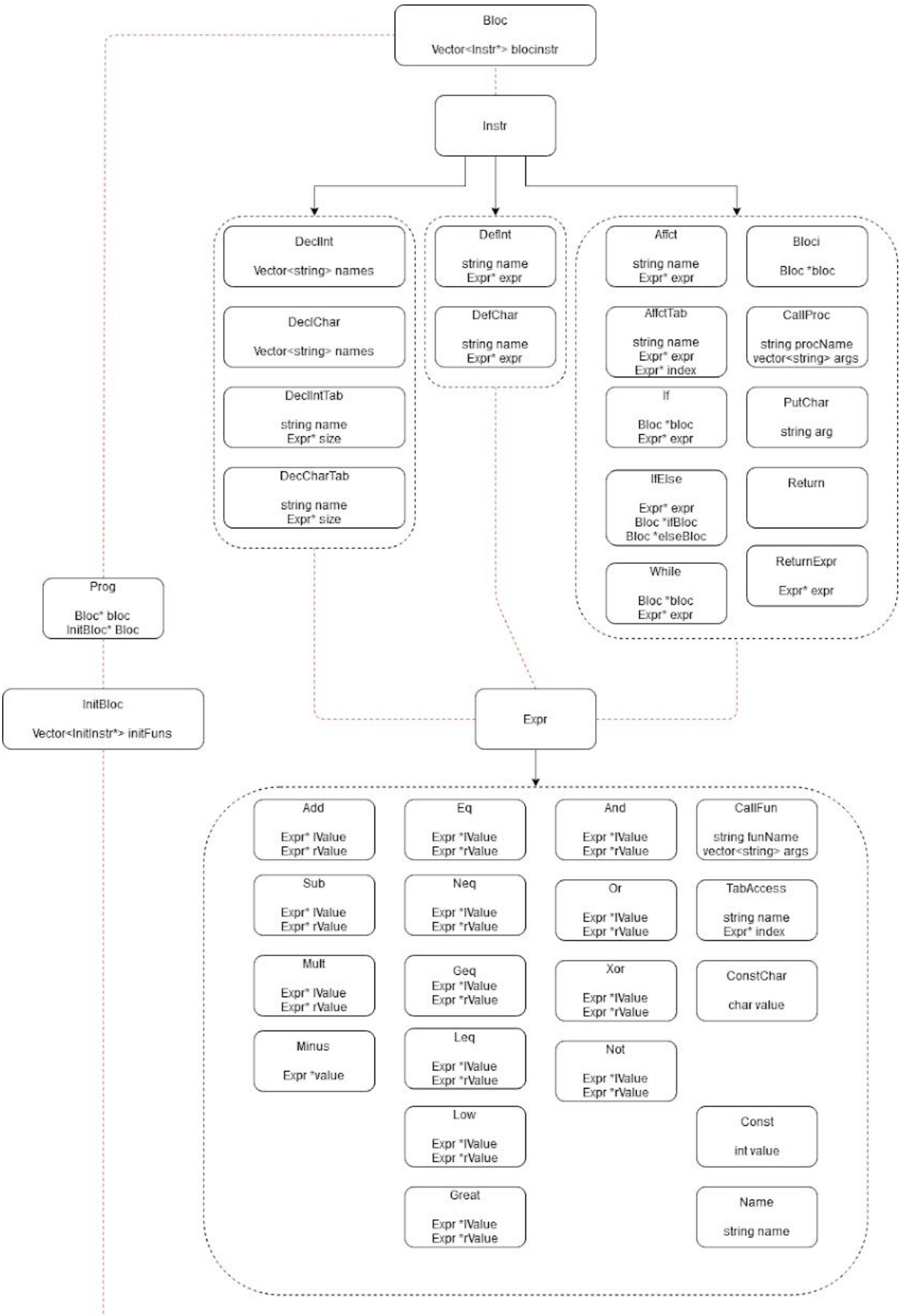
La grammaire suivie par Antlr4 peut être visualisée en cliquant sur ce lien. A partir du fichier grammaire *ifcc.g4*, Antlr4 génère un lexer et un parser, qui permettent de générer l'arbre de dérivation d'un code source à compiler.

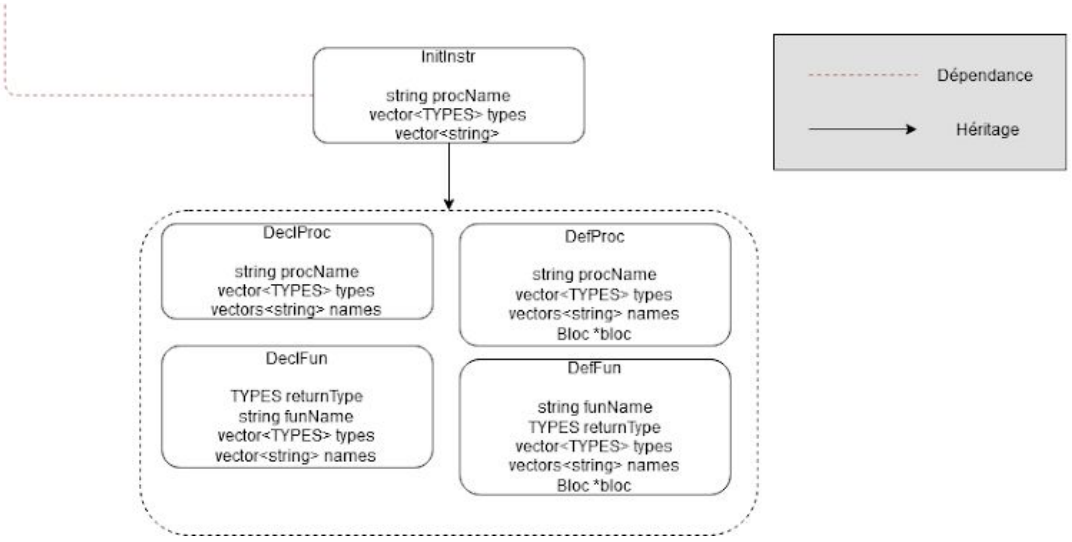
Nous avons ensuite décidé de faire un AST, ce qui nous a permis de mieux séparer le travail dans l'hexanôme entre une équipe qui travaille sur la grammaire et la construction de l'AST, et une autre qui récupère ce travail pour produire l'assembleur.

L'AST est constitué d'une multitude de classes pour former une structure d'arbre comme représenté sur le diagramme ci-dessous.

Nous avons également mis en place quelques optimisations dans l'AST :

- La propagation des constantes, 3 * 4 est par exemple remplacé par 12, y compris en prenant en compte l'associativité et la commutativité, par exemple (1 + a) + (b + 2) est remplacé par a + b + 3
- La neutralité de 0 et 1 respectivement pour l'addition et la multiplication, par exemple 0 + a est remplacé par a
- L'absorbance de 0 pour la multiplication, par exemple 0 * a est remplacé par 0
- L'élimination du code mort après un if, un else un while ou un for, par exemple if(0){...} est supprimé
- La détection de certaines boucles infinies, par exemple while(1){...} va générer un warning si les … ne contiennent pas de return


Bloc
Vector<Instr*> blocinstr
Instr
DeclInt
Vector<string> names
DeclChar
Vector<string> names
DeclIntTab
string name
Expr* size
DecCharTab
string name
Expr* size
DefInt
string name
Expr* expr
DefChar
string name
Expr* expr
Affct
string name
Expr* expr
AffctTab
string name
Expr* expr
Expr* index
If
Bloc *bloc
Expr* expr
IfElse
Expr* expr
Bloc *ifBloc
Bloc *elseBloc
While
Bloc *bloc
Expr* expr
Bloci
Bloc *bloc
CallProc
string procName
vector<string> args
PutChar
string arg
Return
ReturnExpr
Expr* expr
Prog
Bloc* bloc
InitBloc* Bloc
InitBloc
Vector<InitInstr*> initFuns
Expr
Add
Expr* lValue
Expr* rValue
Sub
Expr* lValue
Expr* rValue
Mult
Expr* lValue
Expr* rValue
Minus
Expr *value
Eq
Expr *lValue
Expr *rValue
Neq
Expr *lValue
Expr *rValue
Geq
Expr *lValue
Expr *rValue
Leq
Expr *lValue
Expr *rValue
Low
Expr *lValue
Expr *rValue
Great
Expr *lValue
Expr *rValue
And
Expr *lValue
Expr *rValue
Or
Expr *lValue
Expr *rValue
Xor
Expr *lValue
Expr *rValue
Not
Expr *lValue
Expr *rValue
CallFun
string funName
vector<string> args
TabAccess
string name
Expr* index
ConstChar
char value
Const
int value
Name
string name




# Construction et utilisation de l'IR

## Construction de la représentation intermédiaire

Nous avons décidé d'implémenter une IR, puisque cette représentation du code permettait 'une meilleure maintenabilité du compilateur.

Notre IR contient les classes suivantes :

- CFG : Structure de donnée (graphe) qui stocke les Basic Blocks sous forme de noeuds, qui représente les chemins suivis par un programme lors de son exécution.
- Basic Block : Classe qui représente le bloc qui contient une suite d'instructions jusqu'à un saut.
- IRInstr : Correspond à une instruction précise ayant son sens au niveau de l'assembleur. Une instruction contient notamment un opérateur et un vecteur de string permettant de stocker les différents opérandes de l'instruction.
  *Exemple :*
  *3 + 4 sera représenté sous la forme d'une instruction dont l'opérateur est ADD et dont les opérandes sont <3,4,res> avec res représentant la variable qui stocke le résultat.*
- Type : classe qui stocke des données relatives au type employé (l'offset notamment pour l'assembleur, avec un offset de 4 (bytes) pour le type int et de 1 pour le type char).

L'IR est créé grâce à la fonction *generateIR* appelée sur un AST::Prog* qui est le noeud le plus haut du programme. Cette fonction permet de créer un pointeur de CFG qui pointera toujours vers le CFG actuel, ainsi qu'un vecteur de pointeurs permettant de stocker les pointeurs des différents CFGs. Le pointeur de CFG actuel est donc ajouté à ce vecteur. Puis les noeuds fils de AST::Prog sont visités.

Lors du parcours de ces noeuds fils, les variables temporaires nécessaires sont créées par le CFG actuel (appel de la fonction *create_new_temp_var*) , les variables sont ajoutées à la table des symboles (appel de la fonction *add_to_symbol_table)*, et les instructions sont ajoutées au basic block du CFG actuel ( appel de la fonction *add_IRInstr*).
Lorsque le noeud visité est une fonction, un nouveau pointeur de CFG est créé et ajouté au vecteur de pointeurs de CFG. Le pointeur de CFG actuel est mis à jour.
A la fin du parcours des fils, nous avons la représentation du code sous forme de structure de donnée : l'IR.

### Génération du code assembleur

Afin de générer le code assembleur x86 à partir de notre représentation intermédiaire, il suffit de parcourir le vecteur de pointeurs de CFG créé précédemment, et, appeler la fonction *gen_asm* sur chaque CFG.
Lorsqu'on utilise la méthode gen_asm de la classe CFG, ce dernier parcourt les Basic Blocks en appelant leur méthode du même nom.
Les Basic Blocks vont à leur tour appeler la méthode *gen_asm* sur les instructions qu'ils contiennent.
La méthode *gen_asm* des instructions permettra de créer le code assembleur correspondant à chaque instruction, et de l'écrire dans le flux de string fourni en entrée.

S'il n'y a eu aucune erreur lors des étapes précédentes, le code assembleur écrit sur le flux de string sera mis dans un fichier .s pour ensuite être exécuté. Le nom du fichier .s dépendra du nom du fichier .c compilé. En effet, si nous compilons le fichier programme.c, le fichier programme.s sera généré.

### Gestion des erreurs

L'IR réalise l'analyse syntaxique et envoie une erreur lorsque :

- une variable non définie est utilisée
- une variable est définie deux fois dans le même bloc
- une fonction n'a pas été définie
- une fonction non déclarée est utilisée
- une procédure renvoie une valeur non vide
- une fonction ne renvoie pas de valeur

Lorsqu'une erreur est rencontrée, elle est ajoutée à l'attribut de type Erreur du CFG actuel, puis elle est affichée sur la sortie standard. Le message d'erreur contient le numéro de ligne, le numéro de colonne et la description de l'erreur.
*Exemple :*

*error line 3 column 8 : cannot find the offset, the variable var has not been declared*

Le code assembleur n'est créé et le fichier .s n'est généré que s'il n'y a pas eu d'erreur lors de la construction de l'IR.

# Fonctionnalités implémentées

En plus d'avoir implémenté la grammaire décrite ci-dessus nous avons implémenté :

- La déclaration, la définition et l'affectation de variables de type int et char
    - Il est possible d'affecter une valeur directement lors de la création de cette dernière ( ex : int a = 5; )
    - Nous avons aussi implémenté les variables locales, et donc la portée de ces dernières, au bloc dans lequel elles ont été définies.
- La déclaration, la définition et l'affectation de tableau de int ou de char à taille fixe (ex : int tab[2]; tab[0] = 1;)
- La déclaration et définition de fonctions de type de retour int, char et void (procédures) avec le support de 7 arguments maximum
    - Support de fonctions récursives
- Des structures telles que :
    - La structure conditionnelle if/else ( sans else if )
    - La boucle while
    - La boucle for sur un itérateur de type int ou char
    - Boucle infinie possible
- La déclaration, la définition et l'affectation de tableaux d'entiers à une dimension:
    - Définition avec une taille de valeur constante ( ex : int a[1]={2,3}; )
    - Affectation des valeurs à des indices constants ( ex : a[0] = 3; a[1]=1; )
- Toutes les expressions liées aux opérateurs sur les variables et expressions logiques sauf "/" (donc + (addition), - (soustraction et - unaire), * (multiplication), = (affectation), < (inférieur), > (supérieur), >= (supérieur ou égal), <= (inférieur ou égal), != (non égalité), == (égalité), & (et bit à bit), | (ou bit à bit), ^ (xor bit à bit), !(négation logique))
- Les entrées/sorties avec putchar et getchar
- La déclaration de blocs (et même de blocs dans un bloc)
- L'introduction de commentaires avec // ( sur une ligne ), /* */ ( en bloc )

# Fonctionnalités non-implémentées

Nous avons comme fonctionnalités non-implémentées :

- Les variables globales
- Les struct
- Les pointeurs
- Les includes ( bien que le compilateur ignore volontairement ce qui est écrit après un croisillon (#) )
- Affectation de valeurs dans un tableau en précisant l'indice par une variable (ex : int a[2]; int b= 0; a[b]=3;)

# Annexe - Grammaire

Voici la description de la grammaire implémentée sur Antlr4 pour notre compilateur.

## 1. Symboles terminaux

```
CONST : [0-9]+ ;
CONSTCHAR : '\'' [a-zA-Z_0-9] '\'' ;
NAME : [a-zA-Z_]+[a-zA-Z_0-9]* ; //chiffres lettres underscore et blanc souligne
ou blanc souligne tout seul
COMMENT1 : '/*' .*? '*/' -> skip ;
COMMENT2 : '//' .*? '\n' -> skip ;
DIRECTIVE : '#' .*? '\n' -> skip ;
WS : [ \t\r\n] -> channel(HIDDEN);
```

## 2. Règles de la grammaire

### Types

```
type :
     'int'   #int
    |'char'  #char
;
```

### Structures générales

#### Programme

```
grammar ifcc ;
axiom : prog ;

prog : initbloc 'int' 'main' ‘(‘ ‘)’ ‘{‘ bloc ‘}’ ;
```

#### Bloc

```
bloc : instr*  #blocinstr ;
```

#### Instructions et structures de contrôle

*Les instructions comprennent :*

- *les déclarations, définitions et affectations de variables et de tableaux ;*
- *les structures de contrôle if, while, for*
- *la fonction putchar*

```
instr : 'int' NAME (',' NAME)* ';'                                    #declint
```

```
       |'char' NAME (',' NAME)* ';'
#declchar
       |'int' NAME '[' expr ']' ';'
#declinttab
       |'char' NAME '[' expr ']' ';'
#declchartab
       |'int' NAME '=' expr ';'                                    #defint
       |'char' NAME '=' expr ';'                                   #defchar
       |NAME '=' expr ';'                                          #affexpr
       |NAME '[' expr ']' '=' expr ';'                             #afftab
       |'if' '(' expr ')' '{' bloc '}'                             #ifbloc
       |'if' '(' expr ')' '{' bloc '}' 'else' '{' bloc'}'
#ifelsebloc
       |'while' '(' expr ')' '{' bloc '}'
       #whilebloc
       |'for' '(' initfor? ';' expr? ';' (loopinstr(',' loopinstr)*)? ')'
             '(' bloc '}'                                          #forbloc
       |'{' bloc '}'
#instrbloc
       |NAME '(' (NAME (',' NAME)*)? ')' ';'
#callproc
       |'putchar' '(' NAME ')' ';'                                 #putchar
       |'return' ';'                                               #return
       |'return' expr ';'
#returnexpr
;

initfor :
        'int' NAME (',' NAME)*      #fordeclint
       |'char' NAME (',' NAME)*     #fordeclchar
       |'int' NAME '[' expr ']'     #fordeclinttab
       |'char' NAME '[' expr ']'    #fordeclchartab
       |'int' NAME '=' expr         #fordefint
       |'char' NAME '=' expr        #fordefchar
       |NAME '=' expr               #foraffexpr
       |NAME '[' expr ']' '=' expr  #forafftab
;

loopinstr :
        NAME '=' expr               #loopaff
       |NAME '[' expr ']' '=' expr  #loopafftab
;
```

## Expressions

*Les expressions comprennent :*

- *les expressions parenthésées*
- *les expressions classiques calculatoires (addition, multiplication, etc.)*

- *les expressions de comparaison (inférieur, égal, supérieur ou égal, etc.)*
- *les expressions logiques (non, ou, et, etc.)*
- *les constantes*
- *les noms (de variable etc.)*
- *la fonction getchar*

```
expr : '('expr')'                                  #par
       |'-' expr                                   #minus
       |expr '*' expr                              #mult
       |expr '-' expr                              #sub
       |expr '+' expr                              #add
       |'!' expr                                   #not
       |expr '==' expr                             #eq
       |expr '!=' expr                             #neq
       |expr '<=' expr                             #leq
       |expr '>=' expr                             #geq
       |expr '<' expr                              #low
       |expr '>' expr                              #geat
       |expr '&' expr                              #and
       |expr '^' expr                              #xor
       |expr '|' expr                              #or
       |NAME '(' (NAME (',' NAME)*)? ')'           #callfun
       |'getchar' '(' ')'                          #getchar
       |CONST                                      #const
       |CONSTCHAR                                  #constchar
       |NAME                                       #name
       |NAME '[' expr ']'                          #tabaccess
;
```

## Fonctions et procédures

```
initbloc : initfun*  #blocinit ;

initfun : 'void' NAME '(' (type NAME (',' type NAME)*)? ')' ';'        #declproc
       |type NAME '(' (type NAME (',' type NAME)*)? ')' ';'            #declfun
       |'void' NAME '(' (type NAME (',' type NAME)*)? ')''{' bloc '}'  #defproc
       |type NAME '(' (type NAME (',' type NAME)*)? ')' '{' bloc '}'   #deffun
;
```