



Cours de Compilateur

Rapport de projet

Éric Abouaf, Rodica Militaru
février 2006

1. Introduction:

Le compilateur réalisé au cours de cet exercice réalise les tâches suivantes :

- Analyse syntaxique du programme source en respectant la grammaire de l'exercice (Cf. 2.a)
- Traduction en pseudo-assembleur
- Optimisation en vue de limiter le nombre de garages temporaires

Malheureusement, nous ne sommes pas parvenus à générer un code effectuant de transtypage en fonction des types de variables.

Le compilateur est écrit en C. Le code source est disponible en annexe.

2. Grammaire:

a) Grammaire originale:

- (01): $\langle \text{bloc} \rangle ::= \text{BLOC } \langle \text{decls} \rangle \text{ DEBUT } \langle \text{insts} \rangle \text{ FIN}$
- (02): $\langle \text{decls} \rangle ::= \langle \text{decl} \rangle \mid \langle \text{decls} \rangle ; \langle \text{decl} \rangle$
- (03): $\langle \text{decl} \rangle ::= \langle \text{type} \rangle \langle \text{liste_id} \rangle$
- (04): $\langle \text{type} \rangle ::= \text{ENTIER} \mid \text{FLOTTANT} \mid \text{COMPLEXE}$
- (05): $\langle \text{liste_id} \rangle ::= \langle \text{id} \rangle \mid \langle \text{liste_id} \rangle , \langle \text{id} \rangle$
- (06): $\langle \text{insts} \rangle ::= \langle \text{inst} \rangle \mid \langle \text{insts} \rangle ; \langle \text{inst} \rangle$
- (07): $\langle \text{inst} \rangle ::= \langle \text{var} \rangle := \langle \text{expr} \rangle \mid \langle \text{bloc} \rangle$
- (08): $\langle \text{expr} \rangle ::= \langle \text{terme} \rangle \mid \langle \text{expr} \rangle \langle \text{op_add} \rangle \langle \text{terme} \rangle$
- (09): $\langle \text{op_add} \rangle ::= + \mid -$
- (10): $\langle \text{terme} \rangle ::= \langle \text{fact} \rangle \mid \langle \text{terme} \rangle \langle \text{op_mul} \rangle \langle \text{fact} \rangle$
- (11): $\langle \text{op_mul} \rangle ::= * \mid /$
- (12): $\langle \text{fact} \rangle ::= (\langle \text{expr} \rangle) \mid \langle \text{var} \rangle$
- (13): $\langle \text{var} \rangle ::= \langle \text{id} \rangle$
- (14): $\langle \text{id} \rangle ::= a \mid b \mid \dots \mid z$

Les règles numérotés 2b, 5b, 6b, 8b et 10b sont récursives à gauche. La grammaire n'est donc pas LL(1) et ne peut pas être implémentée telle quelle.

b) Grammaire modifiée:

Voici les règles modifiées :

- (02a): $\langle \text{decls} \rangle ::= \langle \text{decl} \rangle \langle \text{decls1} \rangle$
- (02b): $\langle \text{decls1} \rangle ::= \epsilon$
- (02c): $\langle \text{decls1} \rangle ::= ; \langle \text{decl} \rangle \langle -$
- (05a): $\langle \text{liste_id} \rangle ::= \langle \text{elt} \rangle \langle \text{liste_id1} \rangle$
- (05b): $\langle \text{liste_id1} \rangle ::= \epsilon$
- (05c): $\langle \text{liste_id1} \rangle ::= , \langle \text{elt} \rangle \langle -$
- (05d): $\langle \text{elt} \rangle ::= \langle \text{id} \rangle$

(06a): $\langle \text{insts} \rangle ::= \langle \text{inst} \rangle \langle \text{insts1} \rangle$
 (06b): $\langle \text{insts1} \rangle ::= \epsilon$
 (06c): $\langle \text{insts1} \rangle ::= ; \langle \text{inst} \rangle \langle -$

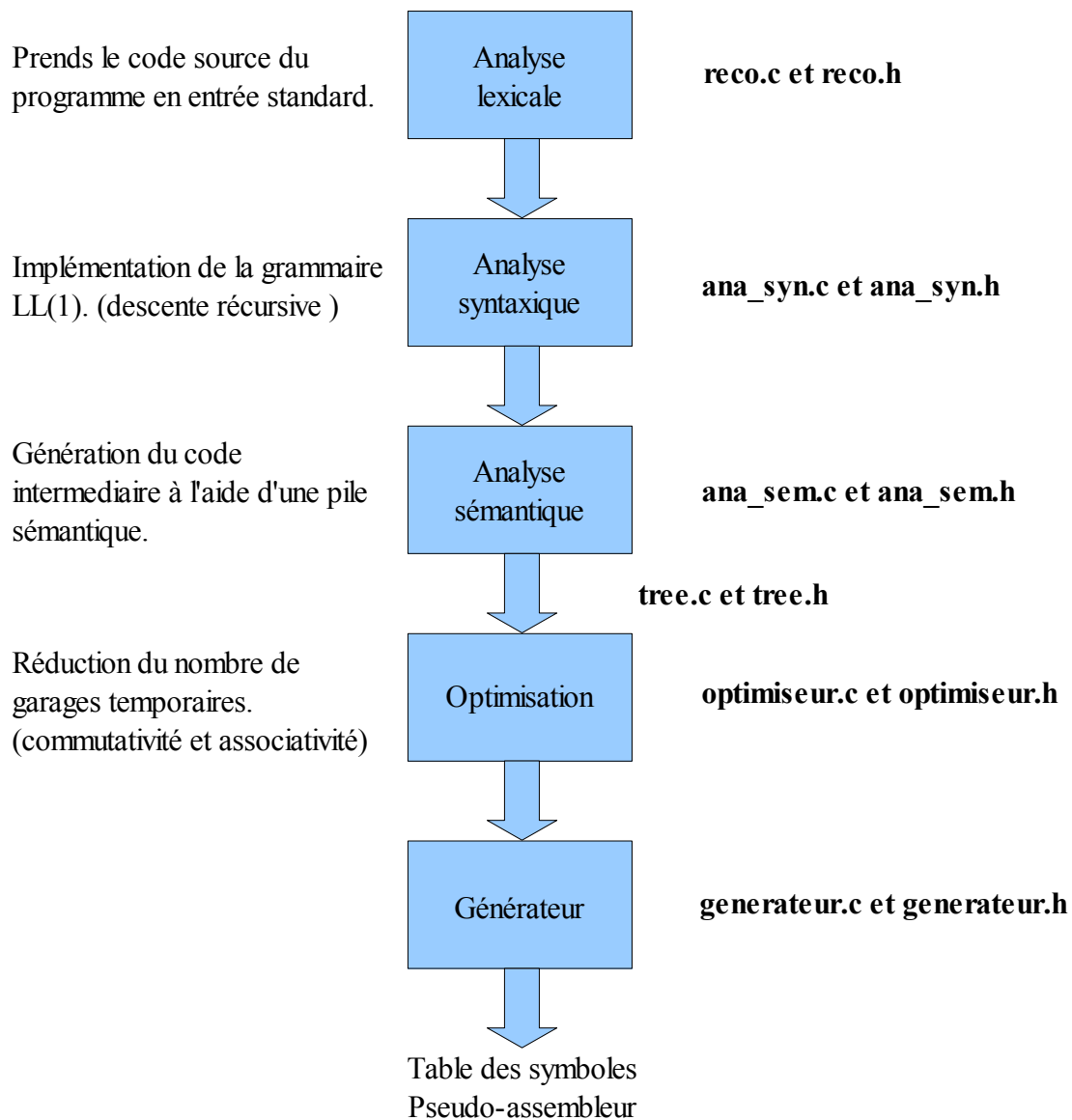
 (08a): $\langle \text{expr} \rangle ::= \langle \text{terme} \rangle \langle \text{expr1} \rangle$
 (08b): $\langle \text{expr1} \rangle ::= \epsilon$
 (08c): $\langle \text{expr1} \rangle ::= \langle \text{op_add} \rangle \langle \text{terme} \rangle \langle -$

 (10a): $\langle \text{terme} \rangle ::= \langle \text{fact} \rangle \langle \text{terme1} \rangle$
 (10b): $\langle \text{terme1} \rangle ::= \epsilon$
 (10c): $\langle \text{terme1} \rangle ::= \langle \text{op_mul} \rangle \langle \text{fact} \rangle \langle -$

La grammaire ainsi modifiée devient donc LL(1) :

tete(02b) = SUIV(02a) = { DEBUT }
 tete(02c) = { ; }
 tete(04a), 04b, 04c disjointes deux à deux
 tete(05b) = suiv(05a) = suiv(03) = tete($\langle \text{decls1} \rangle$) = { DEBUT ; }
 tete(05c) = { , }
 tete(06b) = suiv(06a) = { FIN }
 tete(06c) = { ; }
 tete(07a) = tete($\langle \text{var} \rangle$) = tete($\langle \text{id} \rangle$) = { a-z }
 tete(07b) = tete($\langle \text{bloc} \rangle$) = { BLOC }
 tete(08b) = suiv(08a) = SUIV(07a) = tete($\langle \text{insts1} \rangle$) = { FIN ; }
 tete(08c) = tete($\langle \text{op_add} \rangle$) = { + - }
 tete(08a), 08b disjointes
 tete(10b) = suiv(10a) = tete($\langle \text{expr1} \rangle$) = { FIN ; + - }
 tete(10c) = tete($\langle \text{op_mul} \rangle$) = { * / }
 tete(11a), 11b disjointes
 tete(12a) = { (}
 tete(12b) = tete($\langle \text{var} \rangle$) = tete($\langle \text{id} \rangle$) = { a-z }

3. Organisation du compilateur:



La pile sémantique est programmée à l'aide d'une liste chaînée.

La notation intermédiaire est sous forme d'arbre ou tous les noeuds sont des opérateurs binaires et toutes les feuilles des identificateurs (id) de variables. (`tree.c` et `tree.h`)

4. Exemples et résultats :

a) Exemple 1 :

```

BLOC ENTIER i,k,j;
  FLOTTANT a,l; COMPLEXE z
DEBUT
  a:=j;
  z:=a/l;
  BLOC ENTIER l
  DEBUT
    k:=j - i*l - k;
    i:=a/(k+j)*l-((z-a)/(k*j))
  FIN;
  z:= i - z
FIN

```

<p>Table des Symboles</p> <p>z type:2 taille: 8 l type:0 taille: 4 a type:0 taille: 4 j type:1 taille: 2 k type:1 taille: 2 i type:1 taille: 2</p> <p>LDA j STA a LDA a DIV l STA z</p>	<p>Table des Symboles</p> <p>l type:1 taille: 2 z type:2 taille: 8 l type:0 taille: 4 a type:0 taille: 4 j type:1 taille: 2 k type:1 taille: 2 i type:1 taille: 2</p> <p>LDA i MUL l NEG ADD j SUB k STA k LDA k MUL j STA W1</p>	<p>LDA z SUB a DIV W1 STA W0</p> <p>LDA k ADD j STA W1</p> <p>LDA a DIV W1 MUL l ADD W0 STA i LDA i NEG SUB z STA z</p>
---	---	---

b) Exemple 2 :

```

BLOC ENTIER a,b,c,d,e,f
DEBUT
a:=a-b-c-d-(e-f)*a
FIN

```

<p>Table des Symboles</p> <p>f type:1 taille: 2 e type:1 taille: 2 d type:1 taille: 2 c type:1 taille: 2 b type:1 taille: 2 a type:1 taille: 2</p>	<p>LDA e SUB f SUB a NEG ADD a SUB b SUB c SUB d STA a</p>	
--	--	--

c) Exemple 3 :

BLOC ENTIER a,b,c,d,e,f,g,h,i

DEBUT

$h := g * (((a+b)*c+d)*e) - f + (((g+a)*(b+c)) + (d*e)) * (((f+h)*(i+c)) + (d*e))$

FIN

Table des Symboles i type:1 taille: 2 h type:1 taille: 2 g type:1 taille: 2 f type:1 taille: 2 e type:1 taille: 2 d type:1 taille: 2 c type:1 taille: 2 b type:1 taille: 2 a type:1 taille: 2 LDA a ADD b MUL c ADD d MUL e SUB f MUL g STA W0	LDA d MUL e STA W2 LDA i ADD c STA W3 LDA f ADD h MUL W3 ADD W2 STA W1 LDA d MUL e STA W2	LDA b ADD c STA W3 LDA g ADD a MUL W3 ADD W2 MUL W1 ADD W0 STA h
---	--	---

5. Annexe A: Code source

a) Point d'entrée (main.c) :

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include "reco.h"
#include "ana_syn.h"

int main() {
    for (;;) { /* boucle sans fin */
        init_source();
        if (reco(eof)) break; /* arret programme si texte vide */
        if (setjmp(ret) == 0) { /* 'setjmp' pour retour en cas d'erreur */
            lance_compil(); /* lancement compilation */
            printf("Compilation correcte\n");
            verif_fuites_mem();
        }
        printf("Au revoir !\n");
        return EXIT_SUCCESS;
    }
}
```

b) Analyseur lexical (reco.c et reco.h) :

L'analyseur lexical est constitué par les fichiers reco.c et reco.h fournis lors de l'exercice. La seule modification apportée a consisté à enlever la propriété *static* de la variable `no_ligne` afin de pouvoir afficher le numéro de ligne en cas d'erreur syntaxique.

c) Analyseur syntaxique (ana_syn.c et ana_syn.h) :

ana_syn.h

```
#ifndef _ANA_SYN_H_
#define _ANA_SYN_H_

void lance_compil();

#endif
```

ana_syn.c

```
#include <stdio.h>
#include "reco.h"
#include "ana_syn.h"
#include "ana_sem.h"

extern unsigned int no_ligne; /* J'ai enlevé le static dans reco.c */

static void bloc();
static void decls(); static void decls1();static void decl();
static void type();
```

```

static void liste_id(); static void liste_id1();
static void elt();
static void insts(); static void insts1(); static void inst();
static void expr(); static void expr1();
static void op_add();
static void terme(); static void terme1();
static void op_mul();
static void fact();
static void var();
static void id();

/* Point d'entrée de la compilation */
void lance_compil() { bloc(); }

static void bloc()
{
    sem_bloc_debut();
    if( !recoA("BLOC") )
        printf("Ligne %d: BLOC attendu.\n", no_ligne);
    decls();
    if( !recoA("DEBUT") )
        printf("Ligne %d: DEBUT attendu.\n", no_ligne);
    insts();
    if( !recoA("FIN") )
        printf("Ligne %d: FIN attendu.\n", no_ligne);
    sem_bloc();
}

static void decls() { decl(); decls1(); sem_decls(); }
static void decls1() { while( recoA(";") ) decl(); }
static void decl() { type(); liste_id(); }

static void type()
{
    if( recoA("ENTIER FLOTTANT COMPLEXE") ) sem_type();
    else printf("Ligne %d: type non reconnu\n",no_ligne);
}

static void liste_id() { elt(); liste_id1(); }
static void liste_id1() { while( recoA(",") ) elt(); }
static void elt() { id(); sem_elt(); }
static void insts() { inst(); insts1(); }
static void insts1() { while( recoA(";") ) inst(); }
static void inst()
{
    if( reco("BLOC") ) bloc();
    else
    {
        var();
        if( !recoA(":=") )
            printf("Ligne %d: := attendu.\n",no_ligne);
        expr();

        sem_inst();
    }
}

static void expr() { terme(); expr1(); }
static void expr1()
{

```

```

while( reco("+ -") )
{
    op_add();
    terme();
    sem_expr1();
}
}

static void op_add()
{
    if( !recoA("+ -") )
        printf("Ligne %d: + ou - attendu.\n", no_ligne);
    sem_op_add();
}

static void terme() { fact(); terme1(); }
static void terme1()
{
    while( reco("* /") )
    {
        op_mul(); fact();
        sem_terme1();
    }
}

static void op_mul()
{
    if( !recoA("* /") )
        printf("Ligne %d: * ou / attendu.\n", no_ligne);
    sem_op_mul();
}

static void fact()
{
    if( reco("a b c d e f g h i j k l m n o p q r s t u v w x y z") )
        var();
    else
    {
        if( !recoA("(") )
            printf("Ligne %d: ( attendu.\n",no_ligne);

        expr();

        if( !recoA(")") )
            printf("Ligne %d: ) attendu.\n",no_ligne);
    }
}

static void var() { id(); sem_var(); }
static void id()
{
    if( !recoA("a b c d e f g h i j k l m n o p q r s t u v w x y z") )
        printf("Ligne %d: a ... z attendu\n",no_ligne);
    sem_id();
}
}

```

d) Tables des symboles (symboles.c et symboles.h) :

symboles.h

```
#ifndef _SYMBOLES_H_
#define _SYMBOLES_H_

#include "ana_sem.h" /* pour l'enum des TYPES */

typedef struct SYMBOLE{
    int type;
    int id;
    struct SYMBOLE * parent;
} SYMBOLE;

void symbAjoute(char id, int type);
void symbEnleveBloc(SYMBOLE * limit);
void symbAffiche();
int symbCherche(char id);
#endif
```

symboles.c

```
#include <malloc.h>
#include <stdio.h>

#include "symboles.h"

SYMBOLE * tableSymboles = NULL;

void symbAjoute(char id, int type)
{
    SYMBOLE * n;
    if((n = (SYMBOLE *)malloc(sizeof(SYMBOLE))) == NULL)
        { printf("malloc failed "); exit(1); }

    n->type = type;
    n->id = id;
    n->parent = tableSymboles;
    tableSymboles = n;
}

void symbEnleveBloc(SYMBOLE * limit)
{
    SYMBOLE * i = tableSymboles;
    while( limit != i )
        {
            SYMBOLE * old = i;
            i = i->parent;
            free(old);
        }
    tableSymboles = limit;
}
```

```

void symbAffiche()
{
    SYMBOLE * i = tableSymboles;
    printf("-----\n");
    printf("| Table des Symboles |\n");
    printf("-----\n");
    while( i != NULL )
    {
        char taille;
        switch(i->type)
        {
            case TYPE_ENTIER: taille = 2; break;
            case TYPE_FLOTTANT: taille = 4; break;
            case TYPE_COMPLEXE: taille = 8; break;
            default: taille = 0; break;
        }
        printf("| %c type:%d taille: %d |\n", i->id,i->type, taille);
        i = i->parent;
    }
    printf("-----\n");
}

int symbCherche(char id)
{
    SYMBOLE * i = tableSymboles;
    while( i != NULL )
    {
        if( i->id == id)
            return 0; /* trouve ! */
        i = i->parent;
    }
    return 1;
}

```

e) Analyseur sémantique (ana_sem.c et ana_sem.h):

ana_sem.h

```

#ifndef _ANA_SEM_H_
#define _ANA_SEM_H_

#include "symboles.h"
#include "tree.h"
#include "optimiseur.h"
#include "generateur.h"

enum { SEM_TYPE = 0, SEM_ID, SEM_OP, SEM_SYMBOLE, SEM_NODE };
enum { TYPE_FLOTTANT = 0, TYPE_ENTIER, TYPE_COMPLEXE};
enum { OP_PLUS = 0, OP_MOINS, OP_MUL, OP_DIV };

typedef struct SEM_EL{
    int type;
    char id_value;
    int type_value;
    int op_value;
    struct SYMBOLE * symbole;
    struct NODE * node;
    struct SEM_EL * parent;
} SEM_EL;

```

```

/* Actions sémantiques */
void sem_bloc_debut();
void sem_bloc();
void sem_decls();
void sem_type();
void sem_elt();
void sem_inst();
void sem_expr1();
void sem_op_add();
void sem_term1();
void sem_op_mul();
void sem_var();
void sem_id();
#endif

```

ana_sem.c

```

#include <malloc.h>
#include <stdio.h>

#include "ana_sem.h"

extern int rang_lex; /* Rang du dernier lexeme reconnu */
extern unsigned int no_ligne; /* J'ai enlevé le static dans reco.c */
extern SYMBOLE * tableSymboles;

/* Fonctions statiques */
static void semPushId(char id);
static void semPushOp(int op_type);
static void semPushSymb();
static void semPushNode(NODE * node);

static SEM_EL * semTop(int expected_type);
static SEM_EL * semPop(int expected_type);

/* gestion de la pile sémantique */
SEM_EL * last_el = NULL;

void semPushId(char id)
{
    SEM_EL *n;

    if( (n = (SEM_EL *)malloc(sizeof(SEM_EL))) == NULL )
        { printf("malloc failed "); exit(1); }

    n->type = SEM_ID;
    n->id_value = id;

    n->parent = last_el;
    last_el = n;
}

```

```

void semPushOp(int op_type)
{
    SEM_EL *n;

    if( (n = (SEM_EL *)malloc(sizeof(SEM_EL))) == NULL )
    { printf("malloc failed "); exit(1); }

    n->type = SEM_OP;
    n->op_value = op_type;

    n->parent = last_el;
    last_el = n;
}

void semPushSymb()
{
    SEM_EL *n;

    if( (n = (SEM_EL *)malloc(sizeof(SEM_EL))) == NULL )
    { printf("malloc failed "); exit(1); }

    n->type = SEM_SYMBOLE;
    n->symbole = tableSymboles;
    n->parent = last_el;
    last_el = n;
}

void semPushNode(NODE * node)
{
    SEM_EL *n;
    if( (n = (SEM_EL *)malloc(sizeof(SEM_EL))) == NULL )
    { printf("malloc failed "); exit(1); }
    n->type = SEM_NODE;
    n->node = node;
    n->parent = last_el;
    last_el = n;
}

SEM_EL * semPop(int expected_type)
{
    SEM_EL *r = last_el;
    if( r != NULL )
    {
        last_el = r->parent;
        if( r->type != expected_type && expected_type != SEM_NODE )
        {
            printf("Pile sémantique: mauvais type");
            printf(" (attendu:%d trouve:%d)\n",expected_type, r->type);
        }

        if(expected_type == SEM_NODE && r->type == SEM_ID )
        {
            r->type = SEM_NODE;
            r->node = mk_node_id(r->id_value);
        }
    }
    else
        printf("Plus d'élément sur la pile sémantique !\n");
    return r;
}

```

```

SEM_EL * semTop(int expected_type)
{
    if( last_el != NULL )
    {
        if( last_el->type != expected_type && expected_type != SEM_NODE)
        {
            printf("Pile sémantique: mauvais type");
            printf(" (attendu:%d trouve:%d)\n",expected_type, last_el->type);
        }

        if(expected_type == SEM_NODE && last_el->type == SEM_ID )
            semPushNode( mk_node_id(last_el->id_value) );
    }
    else
    {
        printf("Plus d'élément sur la pile sémantique");
        printf(" semTop type attendu=%d\n",expected_type);
    }

    return last_el;
}

void semDisp()
{
    SEM_EL *r = last_el;

    while(r != NULL)
    {
        switch(r->type)
        {
            case SEM_TYPE:
                switch(r->type_value)
                {
                    case TYPE_FLOTTANT: printf("|FLOTTANT|\n"); break;
                    case TYPE_ENTIER: printf("| ENTIER |\n"); break;
                    case TYPE_COMPLEXE: printf("|COMPLEXE|\n"); break;
                    default: printf("|TYPE ERR|\n"); break;
                }
                break;

            case SEM_ID: printf("| %c  |\n", r->id_value);break;
            case SEM_OP:
                switch(r->op_value)
                {
                    case OP_PLUS: printf("| + |\n"); break;
                    case OP_MOINS: printf("| - |\n"); break;
                    case OP_MUL: printf("| * |\n"); break;
                    case OP_DIV: printf("| / |\n"); break;
                    default: printf("| OP ERR |\n"); break;
                }
                break;
            case SEM_SYMBOLE: printf("| SYMBOLE|\n"); break;
            case SEM_NODE: printf("| NODE |\n"); break;
        }
        r = r->parent;
    }
    printf("-----\n");
}

```

```

/* Routines sémantiques */

void sem_bloc_debut() { semPushSymb(); }

void sem_bloc()
{
    SEM_EL * symb;
    /* On enlève tous les types de la pile sémantique */
    SEM_EL * s = last_el;
    while( s != NULL && s->type == SEM_TYPE)
    {
        SEM_EL * type_el = semPop(SEM_TYPE);
        s = type_el->parent;
        free(type_el);
    }
    /* On restaure la table des symboles */
    symb = semPop(SEM_SYMBOLE);
    symbEnleveBloc(symb->symbole);
    free(symb);
}

void sem_decls() { symbAffiche(); }

void sem_type() /* push le type sur la pile */
{
    SEM_EL *n;
    if( (n = (SEM_EL *) malloc(sizeof(SEM_EL))) == NULL )
    { printf("malloc failed "); exit(1); }

    n->type = SEM_TYPE;
    switch( rang_lex )
    {
        case 0: n->type_value = TYPE_ENTIER; break;
        case 1: n->type_value = TYPE_FLOTTANT; break;
        case 2: n->type_value = TYPE_COMPLEXE; break;
    }
    n->parent = last_el;
    last_el = n;
}

void sem_elt()
{
    SEM_EL * id = semPop(SEM_ID);
    SEM_EL * t = semTop(SEM_TYPE);
    /* On ajoute l'id de l'elt à la table des symboles */
    symbAjoute( id->id_value , t->type_value);
    free(id);
}

void sem_inst() /* Uniquement si c'est pas un bloc ! */
{
    SEM_EL * v;
    SEM_EL * t;
    v = semPop(SEM_NODE);
    t = semPop(SEM_ID);

    optimiseur(v->node );
    genInst(v->node ,0);
    freeTree(v->node);
}

```

```

printf("STA %c\n",t->id_value);

free(t);
free(v);
}

void sem_expr1() /* pop 2 opérandes + opérateur => arbre */
{
SEM_EL *r = semPop(SEM_NODE);
SEM_EL *t = semPop(SEM_OP);
SEM_EL *l = semPop(SEM_NODE);
semPushNode( mk_node_op( t->op_value , l->node, r->node ) );
free(r); free(t); free(l);
}

void sem_op_add() /* push op */
{
switch( rang_lex )
{
case 0: semPushOp(OP_PLUS); break;
case 1: semPushOp(OP_MOINS); break;
}
}

void sem_term1()
{
SEM_EL * r = semPop(SEM_NODE);
SEM_EL * t = semPop(SEM_OP);
SEM_EL * l = semPop(SEM_NODE);
semPushNode( mk_node_op( t->op_value , l->node, r->node ) );
free(r); free(l); free(t);
}

void sem_op_mul()
{
switch( rang_lex )
{
case 0: semPushOp(OP_MUL); break;
case 1: semPushOp(OP_DIV); break;
}
}

void sem_var() /* pop id cherche dans les symboles, push le type et l'id */
{
SEM_EL * semID = semPop(SEM_ID);
if( symbCherche(semID->id_value) != 0 )
printf("Ligne %d: '%c' Variable non déclarée\n",no_ligne,semID->id_value);
semPushId(semID->id_value);
free(semID);
}

void sem_id() { semPushId( (char) ('a'+rang_lex) ); }

```

f) Représentation intermédiaire (tree.c et tree.h):

tree.h

```
#ifndef _TREE_H_
#define _TREE_H_

enum { TYPE_OP, TYPE_ID }; /* types pour les feuilles */

typedef struct NODE{
    int type_node; /* oper ou feuille */

    int op_value;
    char id_value;
    int inv;

    struct NODE * leftValue;
    struct NODE * rightValue;
} NODE;

NODE * mk_node_op(int op_type, NODE *l, NODE *r);
NODE * mk_node_id(char val);

int poidsTree(NODE * node);
void freeTree(NODE * node);
void h_swap(NODE *n);
void v_swap(NODE *n);
#endif
```

tree.c

```
#include <malloc.h>
#include <stdio.h>
#include "tree.h"

enum { OP_PLUS = 0, OP_MOINS, OP_MUL, OP_DIV };

void freeTree(NODE *node)
{
    if( node->leftValue != NULL)
        freeTree(node->leftValue);
    if( node->rightValue != NULL)
        freeTree(node->rightValue);
}

void h_swap(NODE * n)
{
    NODE * temp = n->leftValue;
    n->leftValue = n->rightValue;
    n->rightValue = temp;
}

void v_swap(NODE *n)
{
    NODE * temp = n->rightValue;
    n->rightValue = temp->rightValue;
```

```

temp->rightValue = temp->leftValue;
temp->leftValue = n->leftValue;
n->leftValue = temp;
}

int poidsTree(NODE * node)
{
    if( node->type_node == TYPE_ID )
        return 1;
    else
        return poidsTree(node->leftValue)+poidsTree(node->rightValue)+1;
}

NODE * mk_node_op(int op_type, NODE *l, NODE *r)
{
    NODE *n;
    if((n = (NODE *)malloc(sizeof(NODE))) == NULL)
        { printf("malloc failed "); exit(1); }

    n->type_node = TYPE_OP;
    n->op_value = op_type;
    n->leftValue = l;
    n->rightValue = r;
    return n;
}

NODE *mk_node_id(char val)
{
    NODE *n;
    if((n = (NODE *)malloc(sizeof(NODE))) == NULL)
        { printf("malloc failed "); exit(1); }

    n->type_node = TYPE_ID;
    n->id_value = val;
    n->leftValue = NULL;
    n->rightValue = NULL;
    return(n);
}

```

g) Optimisation (optimiseur.c et optimiseur.h) :

optimiseur.h

```

#ifndef _OPTIMISEUR_H_
#define _OPTIMISEUR_H_

#include "tree.h"

void optimiseur(NODE * node);

#endif

```

optimiseur.c

```
#include "optimiseur.h"

enum { OP_PLUS = 0, OP_MOINS, OP_MUL, OP_DIV };

void optimiseur(NODE * node)
{
    if( node->type_node == TYPE_ID )
        return;

    optimiseur(node->rightValue);
    optimiseur(node->leftValue);

    if( node->op_value == OP_MOINS)
    {
        node->op_value = OP_PLUS;
        node->rightValue->inv = 1;
    }

    if( node->op_value == OP_PLUS || node->op_value == OP_MUL)
    {
        /* Associativité */
        if( node->leftValue->type_node == TYPE_OP &&
            node->leftValue->op_value == node->op_value &&
            poidsTree(node->leftValue->rightValue) < poidsTree(node->rightValue))
        {
            h_swap(node);
            v_swap(node);
            h_swap(node);
            optimiseur(node->leftValue);
            optimiseur(node->rightValue);
        }

        /* Commutativité */
        if( poidsTree(node->rightValue) > poidsTree(node->leftValue) )
            h_swap(node);

        /* Associativité */
        if( node->rightValue->type_node == TYPE_OP &&
            node->rightValue->op_value == node->op_value )
        {
            v_swap(node);
            optimiseur(node->leftValue);
            optimiseur(node->rightValue);
        }
    }
}
```

h) Générateur (generateur.c et generateur.h) :

generateur.h

```
#include "tree.h"

void genOp(int op_value, unsigned char var);
void genInst(NODE * node, int p);
```

generateur.c

```
#include <stdio.h>
#include "generateur.h"

enum { OP_PLUS = 0, OP_MOINS, OP_MUL, OP_DIV };

void genOp(int op_value, unsigned char var)
{
    switch(op_value)
    {
        case OP_PLUS: printf("ADD"); break;
        case OP_MOINS: printf("SUB"); break;
        case OP_DIV: printf("DIV"); break;
        case OP_MUL: printf("MUL"); break;
    }
    if( var >= 'a' && var <= 'z')
        printf(" %c\n", var);
    else
        printf(" W%u\n", var);
}

void genInst(NODE * node, int p)
{
    /* Si on a pas un op: */
    if( node->type_node == TYPE_ID)
    {
        printf("LDA %c\n", node->id_value);
        return;
    }

    /* Si on a une feuille a droite: */
    if(poidsTree(node->rightValue) == 1)
    {
        /* On g n re le code de gauche */
        genInst(node->leftValue,p);

        /*On oper avec la feuille: */
        if( node->rightValue != NULL && node->rightValue->inv == 1)
            node->op_value = OP_MOINS;
        if( node->leftValue != NULL && node->leftValue->inv == 1)
            printf("NEG\n");

        genOp(node->op_value, node->rightValue->id_value );
    }
    else
    {
        /* On g n re le code de droite: */
        genInst(node->rightValue,p+1);
        printf("STA W%d\n\n", p);

        /* On g n re le code de gauche */
        genInst(node->leftValue,p+1);

        /* On fait l'op ration */
        genOp(node->op_value, p);
    }
}
}
```