Rapport de Projet - Compilateur MiniC

Sorbonne Université - École Polytech' Sorbonne
Projet Compilation - Année 2025-2026

Auteurs: Emmanuel Pons, Ali Soukarieh

Table des matières

---

1. Introduction

Ce projet consiste à développer un compilateur pour le langage MiniC, un sous-ensemble du langage C. Le compilateur transforme un programme source MiniC en code assembleur MIPS. Ce rapport documente notre démarche de développement, les choix de conception réalisés, et l'état actuel du projet.

Notre approche de développement a suivi l'ordre naturel du processus de compilation :

1. Analyse lexicale (tokenisation)
2. Analyse syntaxique (construction de l'AST)
3. Analyse sémantique (vérifications contextuelles - Passe 1)
4. Génération de code (Passe 2)

---

2. Analyse Lexicale (lexico.l)

2.1 Démarche de développement

La première étape du projet consistait à implémenter l'analyseur lexical dans le fichier `lexico.l`. Notre démarche a été la suivante :

Étape 1 : Définition des expressions régulières

Nous avons commencé par définir les macros Flex pour les éléments de base du langage :

```
LETTRE          [a-zA-Z]
CHIFFRE         [0-9]
CHIFFRE_NON_NUL [1-9]
IDF             {LETTRE}({LETTRE}|{CHIFFRE}|_)*
ENTIER_DEC      0|{CHIFFRE_NON_NUL}{CHIFFRE}*
LETTRE_HEXA     [a-fA-F]
```

```
ENTIER_HEXA     0x({CHIFFRE}|{LETTRE_HEXA})+
ENTIER          {ENTIER_DEC}|{ENTIER_HEXA}
```

Raisonnement : Nous avons suivi strictement la spécification lexicographique du document de spécifications (section 3). La distinction entre `CHIFFRE_NON_NUL` et `CHIFFRE` permet d'éviter les entiers décimaux commençant par 0 (qui seraient ambigus avec la notation octale en C standard).

Étape 2 : Gestion des chaînes de caractères

```
CHAINE_CAR      [^"\\\n]
CHAINE          \"({CHAINE_CAR}|\\\"|\\n)*\"
```

Raisonnement : La spécification indique que les chaînes peuvent contenir :

- Des caractères imprimables sauf `"` et `\`
- Les séquences d'échappement `\"` et `\n`

Nous avons utilisé une classe de caractères négée `[^"\\\n]` pour exclure les caractères problématiques.

Étape 3 : Gestion des commentaires

```
COMMENTAIRE     "//"[^\n]*\n
```

Les commentaires de style C++ (`//`) sont ignorés jusqu'à la fin de ligne.

2.2 Décisions de conception importantes

Gestion des débordements d'entiers :

```
val = strtol(yytext, &endptr, 0);
if (val < -2147483648 || val > 2147483647) {
    fprintf(stderr, "Error line %d: integer overflow for %s\n", yylineno, yytext);
    exit(1);
}
```

Raisonnement : Conformément à la spécification (section 6.5), les entiers sont codés sur 32 bits. Nous détectons les débordements dès l'analyse lexicale car la conversion caractères -> entier se fait à ce moment.

Mode debug pour les tokens :

```
#if LEX_DEBUG
#define RETURN(token) ({ printf("%s \t\"%s\"\n", #token, yytext); return token; })
#else
#define RETURN(token) ({ return token; })
#endif
```

Raisonnement : Cette macro conditionnelle permet de basculer facilement entre un mode debug (affichage des tokens) et le mode normal. Cela nous a permis de vérifier que les tokens étaient correctement reconnus.

2.3 Vérification

Pour vérifier le bon fonctionnement de l'analyseur lexical, nous avons utilisé le fichier de test `test/Syntaxe/OK/test_lex.c`. En mode `LEX_DEBUG`, le lexer affiche la liste des tokens reconnus, ce qui nous a permis de comparer avec les résultats attendus du TD.

---

3. Analyse Syntaxique (grammar.y)

3.1 Démarche de développement

L'analyseur syntaxique construit l'arbre de syntaxe abstraite (AST) du programme. Notre démarche a suivi les étapes suivantes :

Étape 1 : Déclaration des tokens et des types

```
%token TOK_VOID TOK_INT TOK_BOOL TOK_TRUE TOK_FALSE TOK_IF TOK_DO TOK_WHILE TOK_FOR
%token TOK_PRINT TOK_SEMICOL TOK_COMMA TOK_LPAR TOK_RPAR TOK_LACC TOK_RACC

%token <intval> TOK_INTVAL;
%token <strval> TOK_IDENT TOK_STRING;

%type <ptr> program listdecl listdeclnonnull vardecl ident type listtypedecl decl
maindecl
%type <ptr> listinst listinstnonnull inst block expr listparamprint paramprint
```

Étape 2 : Définition des précédences d'opérateurs

```
%nonassoc TOK_THEN
%nonassoc TOK_ELSE
%right TOK_AFFECT
%left TOK_OR
%left TOK_AND
...
%nonassoc TOK_UMINUS TOK_NOT TOK_BNOT
```

Raisonnement : L'ordre des déclarations de précédence suit exactement la spécification (section 4.1). Le conflit classique "dangling else" est résolu par les pseudo-tokens `TOK_THEN` et `TOK_ELSE`. Le moins unaire (`TOK_UMINUS`) a une précédence plus élevée que le moins binaire.

Étape 3 : Implémentation des règles de grammaire

Nous avons implémenté toutes les règles de la grammaire hors-contexte spécifiée dans la section 4.2 du document. Pour chaque règle, nous construisons le noeud AST correspondant.

3.2 Fonctions de construction de l'arbre

Fonction générique `make_node` :

```c
node_t make_node(node_nature nature, int nops, ...) {
    node_t n = malloc(sizeof(node_s));
    n->nature = nature;
    n->nops = nops;
    n->lineno = yylineno;

    if (nops > 0) {
        n->opr = malloc(nops * sizeof(node_t));
        va_list ap;
        va_start(ap, nops);
        for (int i = 0; i < nops; i++)
            n->opr[i] = va_arg(ap, node_t);
        va_end(ap);
    }
    return n;
}
```

Raisonnement : L'utilisation de `va_list` permet de créer des noeuds avec un nombre variable d'enfants de manière élégante. Le numéro de ligne (`yylineno`) est capturé à la création pour permettre des messages d'erreur précis.

Fonctions spécialisées :

- `make_type(node_type t)` : Crée un noeud NODE_TYPE avec le type spécifié
- `make_ident(const char* s)` : Crée un noeud NODE_IDENT avec le nom de l'identificateur
- `make_int(int64_t v)` : Crée un noeud NODE_INTVAL avec la valeur entière
- `make_bool(bool b)` : Crée un noeud NODE_BOOLVAL avec la valeur booléenne
- `make_string(const char* s)` : Crée un noeud NODE_STRINGVAL avec la chaîne

Fonction `list_append` :

```c
node_t list_append(node_t list, node_t elem) {
    return make_node(NODE_LIST, 2, list, elem);
}
```

Raisonnement : Les listes dans l'AST sont représentées par des noeuds NODE_LIST imbriqués. Cette représentation récursive correspond exactement à la grammaire d'arbres spécifiée (règles 0.4, 0.7, 0.16, etc.).

3.3 Vérification

Pour vérifier la construction de l'AST, nous avons utilisé :

1. Le fichier `test/Syntaxe/OK/test_yacc.c` (même code que le TD)
2. La fonction `dump_tree()` qui génère un fichier DOT visualisable avec xdot

L'arbre généré correspond à celui présenté dans la Figure 2 de la spécification, validant ainsi notre implémentation.

---

4. Fonctions Utilitaires (common.c)

4.1 Analyse des arguments de la ligne de commande

Implementation de `parse_args` :

```
void parse_args(int argc, char **argv) {
    int opt;
    while ((opt = getopt(argc, argv, "bo:t:r:svh")) != -1) {
        switch (opt) {
            case 'b': banner = true; break;
            case 'o': outfile = optarg; break;
            case 't': trace_level = atoi(optarg); break;
            case 'r': max_reg = atoi(optarg); break;
            case 's': stop_after_syntax = true; break;
            case 'v': stop_after_verif = true; break;
            case 'h': help = true; break;
        }
    }
    // Validations...
}
```

Raisonnement : Nous avons utilisé `getopt()` malgré la recommandation de la spécification car elle simplifie grandement le parsing des options. Après analyse approfondie du PDF, nous avons constaté que la spécification n'impose aucune contrainte positionnelle stricte : les options et le fichier d'entrée peuvent être mélangés librement dans n'importe quel ordre. Les seules contraintes sont des règles logiques (exclusivité mutuelle, `-b` seul, etc.) que `getopt()` permet de vérifier sans problème après le parsing. Ainsi, la limitation principale de `getopt()` (perte de l'ordre d'apparition des arguments) n'affecte pas notre implémentation.

Les validations suivent exactement les règles de la section 8.1 :

- `-b` doit être utilisé seul
- `-s` et `-v` sont mutuellement exclusifs
- Un seul fichier d'entrée est autorisé
- Les valeurs de `-t` et `-r` sont validées

4.2 Gestion de la mémoire - `free_nodes`

```
void free_nodes(node_t n) {
    if (n == NULL) return;

    // Libération récursive des enfants
    for (int i = 0; i < n->nops; i++) {
        free_nodes(n->opr[i]);
    }

    // Libération du tableau d'opérandes
    if (n->opr != NULL) free(n->opr);

    // Libération des chaînes allouées
    if (n->ident != NULL) free(n->ident);
    if (n->str != NULL) free(n->str);

    // Libération du noeud lui-même
    free(n);
}
```

Raisonnement : On libère d'abord les enfants, puis le noeud courant. Les champs `ident` et `str` sont des chaînes allouées dynamiquement qui doivent être libérées séparément.

Vérification avec Valgrind :

```
valgrind --leak-check=full ./minicc test.c
```

Les tests Valgrind confirment l'absence de fuites mémoire pour les programmes corrects.

4.3 Affichage de l'arbre - `dump_tree`

Cette fonction génère une représentation DOT de l'AST, permettant la visualisation avec xdot. Elle a été très utile pour le debug, notamment pour vérifier que l'arbre construit correspond bien à la grammaire d'arbres.

---

5. Passe 1 - Vérifications Contextuelles

5.1 Architecture de la passe 1

La passe 1 effectue les vérifications sémantiques et décore l'arbre. Notre implémentation dans `passe_1.c` suit la structure suivante :

```c
void analyse_passe_1(node_t root) {
    node_t globals = root->opr[0];
    node_t mainf   = root->opr[1];

    // Traitement des déclarations globales
    push_global_context();
    decls_list(globals, true);

    // Traitement de la fonction main
    main_decl(mainf);

    pop_context();
}
```

5.2 Gestion des déclarations

Fonction `decls_list` :

Parcourt récursivement les listes de déclarations (NODE_LIST ou NODE_DECLS).

Fonction `decl_list` :

```c
void decl_list(node_t decl, node_type type, bool is_global) {
    // ...
    offset = env_add_element(ident_node->ident, ident_node);

    if (type == TYPE_VOID)
        error_rule(ident_node, "1.8", "Variable '%s' cannot be of type void",
ident_node->ident);
```

```
    if (offset == -1)
        error_rule(ident_node, "1.11", "Variable '%s' already declared", ident_node->ident);

    // Vérification de l'initialisation pour les variables globales
    if (decl->nops == 2 && is_global) {
        if (!(decl->opr[1]->nature == NODE_INTVAL || decl->opr[1]->nature == NODE_BOOLVAL))
            error_rule(..., "1.12", "Expression are not allowed...");
        if (decl->opr[1]->type != type)
            error_rule(..., "1.12", "Type mismatch...");
    }
    // ...
}
```

Vérifications implémentées :

- Règle 1.8 : Le type `void` ne peut pas être utilisé pour les variables
- Règle 1.11 : Une variable ne peut pas être déclarée deux fois dans le même contexte
- Règle 1.12 : Les variables globales ne peuvent être initialisées qu'avec des littéraux
- Règle 1.12 : Le type de l'expression d'initialisation doit correspondre au type déclaré

5.3 Vérification de la fonction main

```
void main_decl(node_t func_node) {
    reset_env_current_offset();

    node_t type = func_node->opr[0];
    node_t name = func_node->opr[1];
    node_t block = func_node->opr[2];

    // Règle 1.5 : nom doit être "main"
    if (strcmp(name->ident, "main") != 0)
        error_rule(name, "1.4", "The main function must be named 'main'");

    // Règle 1.5 : type de retour doit être void
    if (type->type != TYPE_VOID)
        error_rule(type, "1.4", "The main function must return 'void'");

    block_decl(block);
    func_node->offset = get_env_current_offset();
}
```

5.4 Gestion de l'environnement

Nous utilisons les fonctions fournies par la bibliothèque `miniccutils` :

- `push_global_context()` : Crée le contexte global
- `push_context()` : Empile un nouveau contexte (pour les blocs)
- `pop_context()` : Dépile le contexte courant
- `env_add_element()` : Ajoute une variable au contexte courant

- `get_decl_node()` : Recherche une variable dans l'environnement
- `get_env_current_offset()` : Récupère l'offset courant pour l'allocation en pile

5.5 Traitement des expressions et instructions

Le parcours de l'arbre pour les vérifications sémantiques repose sur quatre fonctions principales qui s'appellent mutuellement de manière récursive :

```
analyse_passe_1(root)
│
├── decls_list(globals, true)              // Déclarations globales
│
└── main_decl(mainf)
    └── block_decl(block)
        ├── decls_list(block->opr[0], false)   // Déclarations locales
        │   └── expr_list_processing()          // Expressions d'init
        │       └── expr_processing()
        │
        └── instr_list_processing(block->opr[1]) // Instructions
            └── instr_processing()
                ├── expr_processing()            // Conditions
                ├── instr_processing()           // Récursif
                ├── block_decl()                 // Blocs imbriqués
                └── print_list_processing()
```

Parcours des listes (NODE_LIST) :

Les fonctions `expr_list_processing` et `instr_list_processing` utilisent le même pattern récursif pour parcourir les structures NODE_LIST sans rien oublier :

```
void instr_list_processing(node_t instr) {
    if (instr == NULL) return;

    if (instr->nature == NODE_LIST) {
        instr_list_processing(instr->opr[0]);  // Enfant gauche
        instr_list_processing(instr->opr[1]);  // Enfant droit
    } else {
        instr_processing(instr);               // Traitement effectif
    }
}
```

Traitement des expressions (`expr_processing`) :

Cette fonction vérifie les types et décore l'arbre via un `switch` sur la nature du noeud :

Catégorie	Noeuds	Opérandes	Résultat
Arithmétique	PLUS, MINUS, MUL, DIV, MOD	int, int	int
Comparaison	LT, GT, LE, GE	int, int	bool
Égalité	EQ, NE	même type	bool

Catégorie	Noeuds	Opérandes	Résultat
Logique	AND, OR	bool, bool	bool
Bit-a-bit	BAND, BOR, BXOR, SLL, SRA, SRL	int, int	int
Unaire logique	NOT	bool	bool
Unaire arithmétique	UMINUS, BNOT	int	int

Pour les identificateurs (NODE_IDENT), on recherche la déclaration avec `get_decl_node()` et on décore le noeud avec le type, l'offset et le pointeur vers la déclaration.

Pour l'affectation (NODE_AFFECT), on vérifie que l'opérande gauche est un identificateur et que les types correspondent.

Traitement des instructions (`instr_processing`) :

Instruction	Verification	Action recursive
IF	Condition doit être bool (1.18)	`instr_processing` sur then/else
WHILE	Condition doit être bool (1.20)	`instr_processing` sur le corps
FOR	Condition doit être bool (1.21)	`expr_processing` sur init/incr, `instr_processing` sur corps
DOWHILE	Condition doit être bool (1.22)	`instr_processing` sur le corps
BLOCK	-	`block_decl` (nouveau contexte)
PRINT	-	`print_list_processing` sur les paramètres
Default	-	`expr_list_processing` (expression isolée)

Pourquoi rien n'est oublié :

1. Les listes sont parcourues récursivement sur les deux enfants
2. `instr_processing` traite tous les types d'instructions et appelle récursivement les fonctions appropriées
3. `expr_processing` traite tous les types d'expressions et descend dans les opérandes
4. Les blocs imbriqués déclenchent `block_decl` qui crée un nouveau contexte et relance le parcours

5.7 Traitement de l'instruction PRINT

```
void print_processing(node_t print_node){
    if (print_node == NULL) return;

    if (print_node->nature == NODE_IDENT){
        node_t ident_node = get_decl_node(print_node->ident);
        if (ident_node == NULL) {
            error_rule(print_node, "1.61", "Variable '%s' not declared", print_node-
>ident);
        }
        print_node->type = ident_node->type;
        print_node->offset = ident_node->offset;
        print_node->decl_node = ident_node;
        print_node->global_decl = ident_node->global_decl;
    }
}
```

Raisonnement : Les paramètres de `print` peuvent être des chaînes ou des identificateurs. Pour les identificateurs, on vérifie qu'ils sont déclarés et on décore le noeud comme pour les autres utilisations de variables.

---

6. Passe 2 - Génération de Code MIPS

6.1 Architecture de la passe 2

La passe 2 génère le code assembleur MIPS à partir de l'AST décoré. Notre implémentation dans `passe_2.c` suit la structure suivante :

```
void gen_code_passe_2(node_t root) {
    collect_strings(root);

    set_max_registers(get_num_registers());
    reset_temporary_max_offset();

    gen_data_section(root);

    if (root->opr[1] != NULL) {
        gen_text_section(root->opr[1]);
    }
}
```

6.2 Problème initial : allocation de registres

Notre première implémentation de la génération de code utilisait une approche naïve pour l'allocation de registres. Nous appelions `allocate_reg()` dans les noeuds feuilles (NODE_INTVAL, NODE_BOOLVAL, NODE_IDENT), ce qui produisait des numéros de registres décalés par rapport au compilateur de référence.

Exemple du problème :

```
Notre sortie:     ori $9, $0, 42
Référence:        ori $8, $0, 42
```

Après analyse, nous avons découvert que la fonction `get_current_reg()` de la bibliothèque retourne le registre "courant" (initialement $8), et que `allocate_reg()` avance le compteur vers le registre suivant.

6.3 Solution : pattern correct d'allocation

La solution est décrite dans le PDF de spécification. Le pattern correct est :

Pour les noeuds feuilles :

- Utiliser directement `get_current_reg()` pour obtenir le registre
- Ne PAS appeler `allocate_reg()` - c'est l'appelant qui gère l'allocation

Pour les opérations binaires :

```
gen_expr(expr->opr[0]);           // Résultat dans get_current_reg()
reg_left = get_current_reg();     // Sauvegarder le registre
spilled = !reg_available();       // Vérifier si spilling nécessaire
if (spilled) {
    push_temporary(reg_left);     // Sauvegarder sur la pile
}
allocate_reg();                   // Reserver un registre pour l'operande droit
gen_expr(expr->opr[1]);           // Résultat dans get_current_reg()
reg_right = get_current_reg();
// ... effectuer l'opération ...
release_reg();                    // Libérer le registre de l'opérande droit
```

Après avoir appliqué ce pattern, notre sortie correspond exactement à celle du compilateur de référence.

6.4 Génération de la section .data

```
static void gen_data_section(node_t root) {
    create_data_sec_inst();

    if (root->opr[0] != NULL) {
        gen_global_decls(root->opr[0]);
    }

    int32_t num_strings = get_global_strings_number();
    for (int32_t i = 0; i < num_strings; i++) {
        create_asciiz_inst(NULL, get_global_string(i));
    }
}
```

La section .data contient :

- Les variables globales (avec leur valeur initiale ou 0)
- Les chaînes de caractères collectées dans l'arbre

6.5 Génération des expressions

Chaque type d'expression est traité dans un switch :

Littéraux entiers et booléens :

```
case NODE_INTVAL:
case NODE_BOOLVAL:
    reg = get_current_reg();
    if (expr->value >= 0 && expr->value <= 0xFFFF) {
        create_ori_inst(reg, get_r0(), (int32_t)(expr->value & 0xFFFF));
    } else {
        create_lui_inst(reg, (int32_t)((expr->value >> 16) & 0xFFFF));
        create_ori_inst(reg, reg, (int32_t)(expr->value & 0xFFFF));
    }
    break;
```

Chargement de variables :

```
case NODE_IDENT:
    if (decl != NULL && decl->global_decl) {
        create_lui_inst(reg, 0x1001);
        create_lw_inst(reg, decl->offset, reg);
    } else {
        create_lw_inst(reg, expr->offset, get_stack_reg());
    }
    break;
```

6.6 Gestion du spilling de registres

Quand tous les registres temporaires sont utilisés, on sauvegarde sur la pile :

```
case NODE_PLUS:
    gen_expr(expr->opr[0]);
    reg_left = get_current_reg();
    spilled = !reg_available();
    if (spilled) {
        push_temporary(reg_left);
    }
    allocate_reg();
    gen_expr(expr->opr[1]);
    reg_right = get_current_reg();
    if (spilled) {
        pop_temporary(get_restore_reg());
        create_addu_inst(reg_right, get_restore_reg(), reg_right);
    } else {
        create_addu_inst(reg_left, reg_left, reg_right);
        release_reg();
    }
    break;
```

6.7 Structures de contrôle

Instruction IF :

```
static void gen_if(node_t node) {
    int32_t label_else = get_new_label();

    gen_expr(node->opr[0]);
    create_beq_inst(get_current_reg(), get_r0(), label_else);

    gen_instr(node->opr[1]);

    if (node->opr[2] != NULL) {
        int32_t label_end = get_new_label();
        create_j_inst(label_end);
        create_label_inst(label_else);
        gen_instr(node->opr[2]);
```

```
        create_label_inst(label_end);
    } else {
        create_label_inst(label_else);
    }
}
```

Instruction WHILE :

```
static void gen_while(node_t node) {
    int32_t label_start = get_new_label();
    int32_t label_end = get_new_label();

    create_label_inst(label_start);
    gen_expr(node->opr[0]);
    create_beq_inst(get_current_reg(), get_r0(), label_end);
    gen_instr(node->opr[1]);
    create_j_inst(label_start);
    create_label_inst(label_end);
}
```

6.8 Gestion de la pile

Nous utilisons les fonctions de la bibliothèque pour gérer la pile :

```
static void gen_text_section(node_t func) {
    create_text_sec_inst();
    create_label_str_inst("main");

    set_temporary_start_offset(func->offset);
    create_stack_allocation_inst();

    if (func->opr[2] != NULL) {
        gen_block(func->opr[2]);
    }

    int32_t stack_size = get_temporary_max_offset();
    if (func->offset > stack_size) {
        stack_size = func->offset;
    }
    create_stack_deallocation_inst(stack_size);
    create_ori_inst(2, get_r0(), 0xa);
    create_syscall_inst();
}
```

6.9 Division par zéro

Pour les opérations DIV et MOD, nous ajoutons une vérification :

```
create_div_inst(reg_left, reg_right);
create_teq_inst(reg_right, get_r0());  // Trap si diviseur = 0
```

```
create_mflo_inst(reg_left);            // ou mfhi pour MOD
```

6.10 Résultats des tests

Catégorie	Résultat
Tests OK (génération)	14/14 pass
Tests KO (division par zéro)	3/3 pass
Total	**17/17 pass**

---

7. Tests Réalisés

Nous avons écrit 57 tests pour valider notre compilateur. Les tests fournis initialement (`test_lex.c`, `test_yacc.c`) ont été réorganisés et complétés.

7.1 Tests Syntaxe (Tests/Syntaxe/)

Tests KO (erreurs de syntaxe) :

Fichier	Description
test_nok1.c	Déclaration sans point-virgule
test_nok2.c	Parenthèse manquante
test_nok3.c	Accolade manquante
test_nok4.c	Mot-clé mal orthographié
test_nok5.c	Expression mal formée

7.2 Tests Verif - Passe 1 (Tests/Verif/)

Tests OK (15 tests) : Programmes valides testant les déclarations, expressions, et structures de contrôle.

Tests KO (20 tests) :

Fichier	Regle	Description
test-ko1-main-misnamed	1.4	Fonction principale mal nommée
test-ko2-main-not-void	1.4	Type de retour de main != void
test-ko3-void-var	1.8	Variable de type void
test-ko4-double-decl	1.11	Double déclaration
test-ko5-undecl-var	1.61	Variable non déclarée
test-ko6-if-cond-not-bool	1.18	Condition if non booléenne
test-ko7-while-cond-not-bool	1.20	Condition while non booléenne
test-ko8-for-cond-not-bool	1.21	Condition for non booléenne
test-ko9-dowhile-cond-not-bool	1.22	Condition do-while non booléenne

Fichier	Regle	Description
test-ko10-assign-typemismatched	1.32	Types incompatibles dans affectation
test-ko11-arith-with-bool	1.30	Opérateur arithmétique avec bool
test-ko12-logic-with-int	1.30	Opérateur logique avec int
test-ko13-bitwise-bool	1.30	Opérateur bit-à-bit avec bool
test-ko14-not-with-int	1.31	NOT logique avec int
test-ko15-bnot-with-bool	1.31	NOT bit-à-bit avec bool
test-ko16-==-bool-and-int	1.30	Comparaison bool/int
test-ko17-init-local-not-same-type	1.13	Init locale type incompatible
test-ko18-init-glob-expr	1.12	Init globale avec expression
test-ko19-init-glob-not-same-type	1.12	Init globale type incompatible
test-ko20-double-decl-inner-block	1.11	Double déclaration dans bloc imbriqué

7.3 Tests Gencode - Passe 2 (Tests/Gencode/)

Tests OK (14 tests) :

Fichier	Description
test_gencode_01_simple	Affichage simple
test_gencode_02_arithmetic	Operations +, -, *, /, %
test_gencode_03_comparison	Opérateurs <, >, <=, >=, ==, !=
test_gencode_04_logical	Opérateurs &&, ||, !
test_gencode_05_bitwise	Opérateurs &, |, ^, ~, <<, >>
test_gencode_06_if	Structure if simple
test_gencode_07_ifelse	Structure if-else
test_gencode_08_while	Boucle while
test_gencode_09_for	Boucle for
test_gencode_10_dowhile	Boucle do-while
test_gencode_11_global	Variables globales
test_gencode_12_nested	Blocs imbriqués
test_gencode_13_unary	Opérateurs unaires
test_gencode_14_assign	Affectations

Tests KO (3 tests - erreurs runtime) :

Fichier	Description
test_gencode_ko_01_divzero	Division par zéro
test_gencode_ko_02_modzero	Modulo par zéro
test_gencode_ko_03_divzero_expr	Division par zéro dans expression

---

8. Script de Tests (run_tests.sh)

Le script `run_tests.sh` permet d'exécuter les tests automatiquement avec des flags pour sélectionner les catégories.

Note : Ce script a été généré avec l'aide de l'IA.

```
./run_tests.sh [OPTIONS]
```

Flag	Description
`-s`	Tests Syntaxe (analyse syntaxique)
`-v`	Tests Verif (passe 1 - vérifications sémantiques)
`-g`	Tests Gencode (passe 2 - génération de code)
`-a`	Tous les tests
`-h`	Aide

Exemples :

```
./run_tests.sh -a        # Exécuter tous les tests (61 tests)
./run_tests.sh -g        # Seulement les tests Gencode
./run_tests.sh -s -v     # Tests Syntaxe et Verif
```

Résultats actuels : 61/61 tests passent.