



Hexanome - 11

PLD Compilo

Développement d'une chaîne complète de compilation

Sami TAIDER

Riad DALAOUI

Rayan HANADER

Samuel LOUVET

Youssef CHAOUKI

Antoine AUBUT

Djalil CHIKHI



Introduction



Le but : développer un compilateur fonctionnel, de bout en bout, pour
un sous-ensemble du langage C.

Ce compilateur doit analyser un programme source, vérifier sa
validité, et produire un code assembleur, ensuite assemblé avec GCC.



Présentation de l'équipe

Sami TAIDER

Chef de Projet

Rayan HANADER

Responsable Documentation et
Soutenance

Djalil CHIKHI

Architecte Logiciel

Antoine AUBUT

Architecte Logiciel

Riad DALAOUI

Architecte Logiciel

Youssef CHAOUKI

Responsable Qualité

Samuel LOUVET

Responsable Qualité

Mon rôle

Git & Workflow

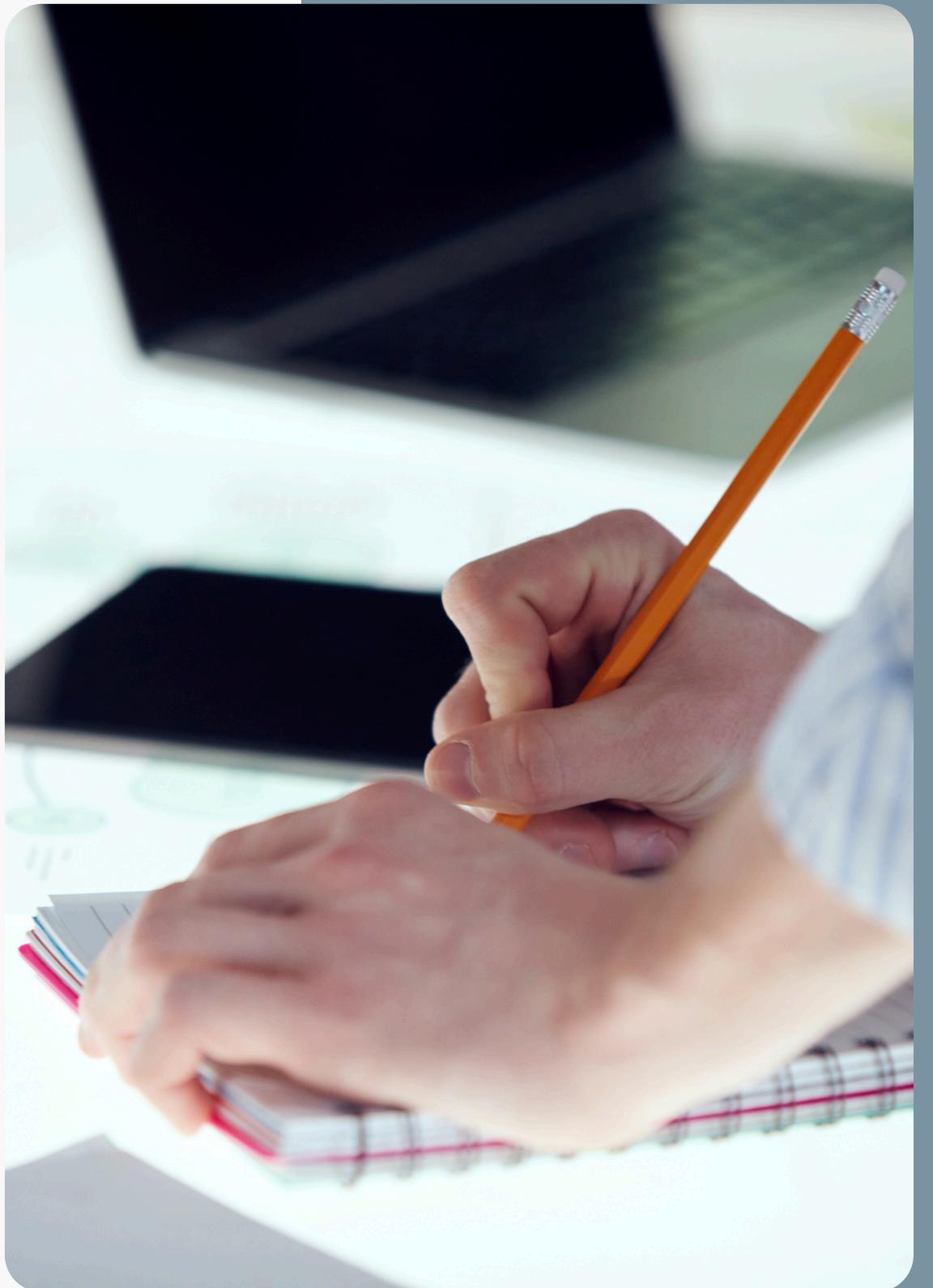
- Structuration du repo
- Suivi des merges

Code Review

- Relecture systématique des pull requests
- Vérification qualité + style
- Suivi des contributions

Organisation Agile

- Sprints de 1 séance
- Objectifs clairs à chaque itération
- Bilan et ajustement à la fin de chaque sprint



Tour d'horizon des fonctionnalités du compilateur

- Partie 1 : Analyse sémantique et gestion des symboles
 - Partie 2 : Parcours de l'AST et production du code exécutable

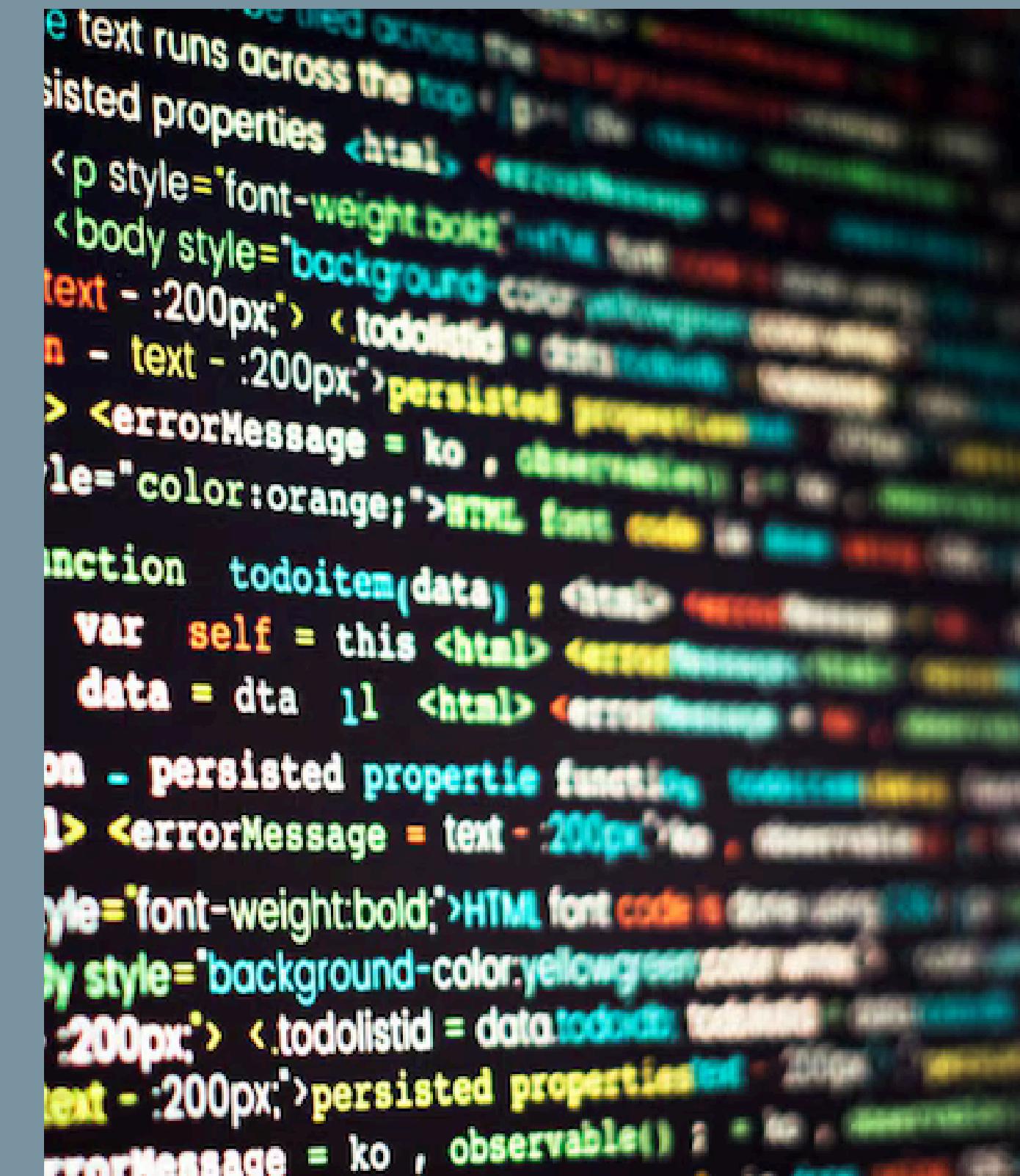
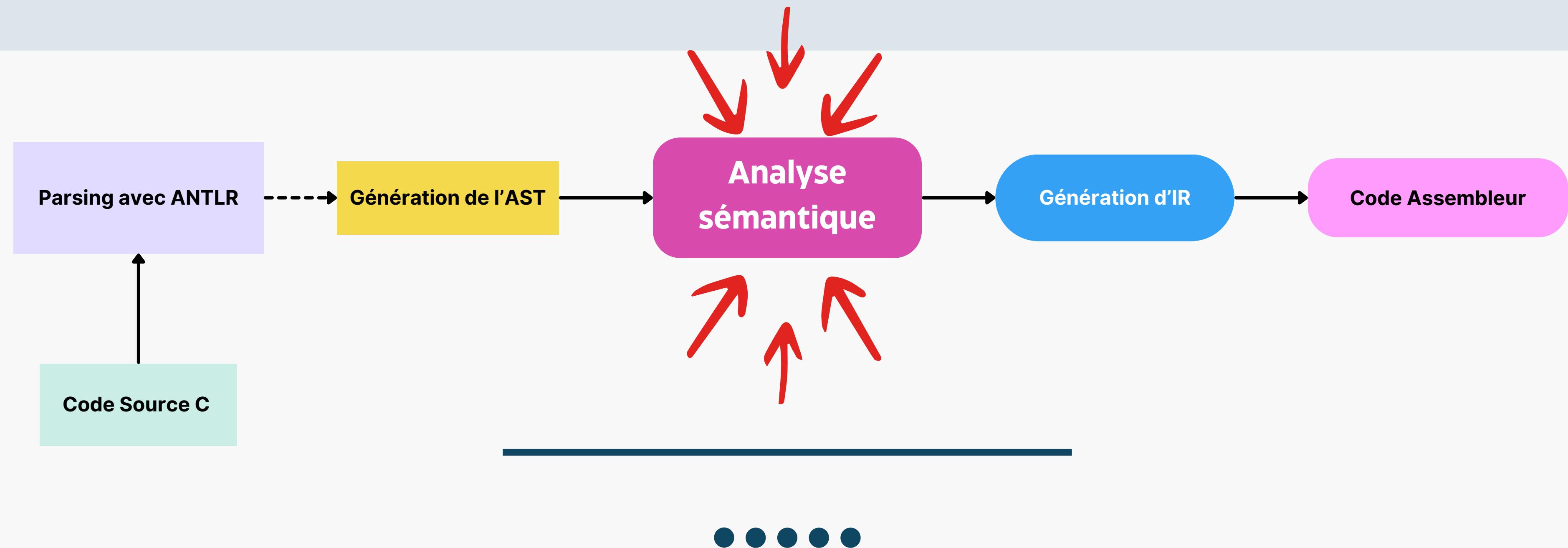


Schéma Global de la Compilation



CodeGenVisitor.cpp

```
bool CodeGenVisitor::addSymbolToSymbolTableFromContext(ParserRuleContext *ctx,
                                                       const string &id, Type type)
{
    bool result = curCfg->add_symbol(id, type, ctx->getStart()->getLine());
    if (!result)
    {
        string error = "The variable " + id + " has already been declared";
        VisitorErrorListener::addError(ctx, error, ErrorType::Error);
    }
    return result;
}
```

IR.cpp

```
bool CFG::add_symbol(string id, Type t, int line)
{
    if (symbolTables.front().count(id))
    {
        return false;
    }
    shared_ptr<Symbol> newSymbol = make_shared<Symbol>(t, id, line);
    unsigned int sz = getSize(t);
    // This expression handles stack alignment
    newSymbol->offset = (nextFreeSymbolIndex + 2 * (sz - 1)) / sz * sz;
    nextFreeSymbolIndex += sz;
    symbolTables.front()[id] = newSymbol;

    return true;
}
```

La Table des Symboles

- Enregistre toutes les déclarations de variables rencontrées dans le code source
- addSymbolToSymbolTableFromContext() assure que chaque variable est déclarée une seule fois
- add_symbol() assigne l'offset correspondant au type de la variable

Gestion et Vérification des Types

	ENTIER	CHAR	VOID
Octets en mémoire	4	1	0

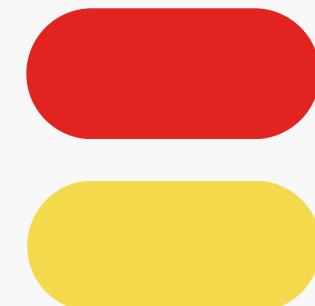
```
unsigned int getSize(Type t) {  
    switch (t) {  
        case Type::INT:  
            return 4;  
        case Type::CHAR:  
            return 1;  
        case Type::VOID:  
            return 0;  
    }  
    return 0;  
}
```

Détection d'Erreurs et Gestion des Avertissements

- Classe VisitorErrorListener créée spécialement pour gérer toutes les erreurs rencontrées

Permet la distinction entre des Warning et des Erreurs avec message personnalisé

```
void VisitorErrorListener::addError(const string &message,  
                                     ErrorType errorType)  
{  
    switch (errorType)  
    {  
        case ErrorType::Error:  
            cerr << "Error: ";  
            mHasError = true;  
            break;  
        case ErrorType::Warning:  
            cerr << "Warning: ";  
            break;  
    }  
  
    cerr << message << endl;  
}
```



Erreur

Warning



	Type	Description
Variable		Impossible de créer une variable de type void
Variable		La variable a déjà été déclarée
Variable		La variable déclarée n'a jamais été utilisée
Fonction		Une fonction non void doit retourner une valeur
Fonction		Une fonction void ne doit rien retourner
Fonction		La fonction n'a pas été déclarée
Fonction		Le nombre de paramètre lors de l'appel ne correspond pas à la définition de la fonction
Fonction		Un paramètre avec ce nom a déjà été déclaré

Parcours de l'AST et production du code exécutable

CodeGenVisitor.cpp

```
bool CodeGenVisitor::addSymbolToSymbolTableFromContext(antlr4::ParserRuleContext *ctx,
                                                       const string &id, Type type)

{
    bool result = curCfg->add_symbol(id, type, ctx->getStart()->getLine());
    if (!result)
    {
        string error = "The variable " + id + " has already been declared";
        VisitorErrorListener::addError(ctx, error, ErrorType::Error);
    }
    return result;
}
```

```
antlr4::Any CodeGenVisitor::visitProg(ifccParser::ProgContext *ctx)
{
    for (ifccParser::FuncContext *func : ctx->func())
    {
        Type type;
        if (func->TYPE(0)->toString() == "int")
        {
            type = Type::INT;
        }
        else if (func->TYPE(0)->toString() == "char")
        {
            type = Type::CHAR;
        }
        else if (func->TYPE(0)->toString() == "void")
        {
            type = Type::VOID;
        }
        int argCount = func->ID().size() - 1;
        curCfg =
            make_shared<CFG>(type, func->ID(0)->toString(), argCount, this);
        cfgList.push_back(curCfg);
        functions[func->ID(0)->toString()] = curCfg;
        visit(func);
        curCfg->pop_table();
    }

    if (VisitorErrorListener::hasError())
    {
        exit(1);
    }

    cout << assembly.str();

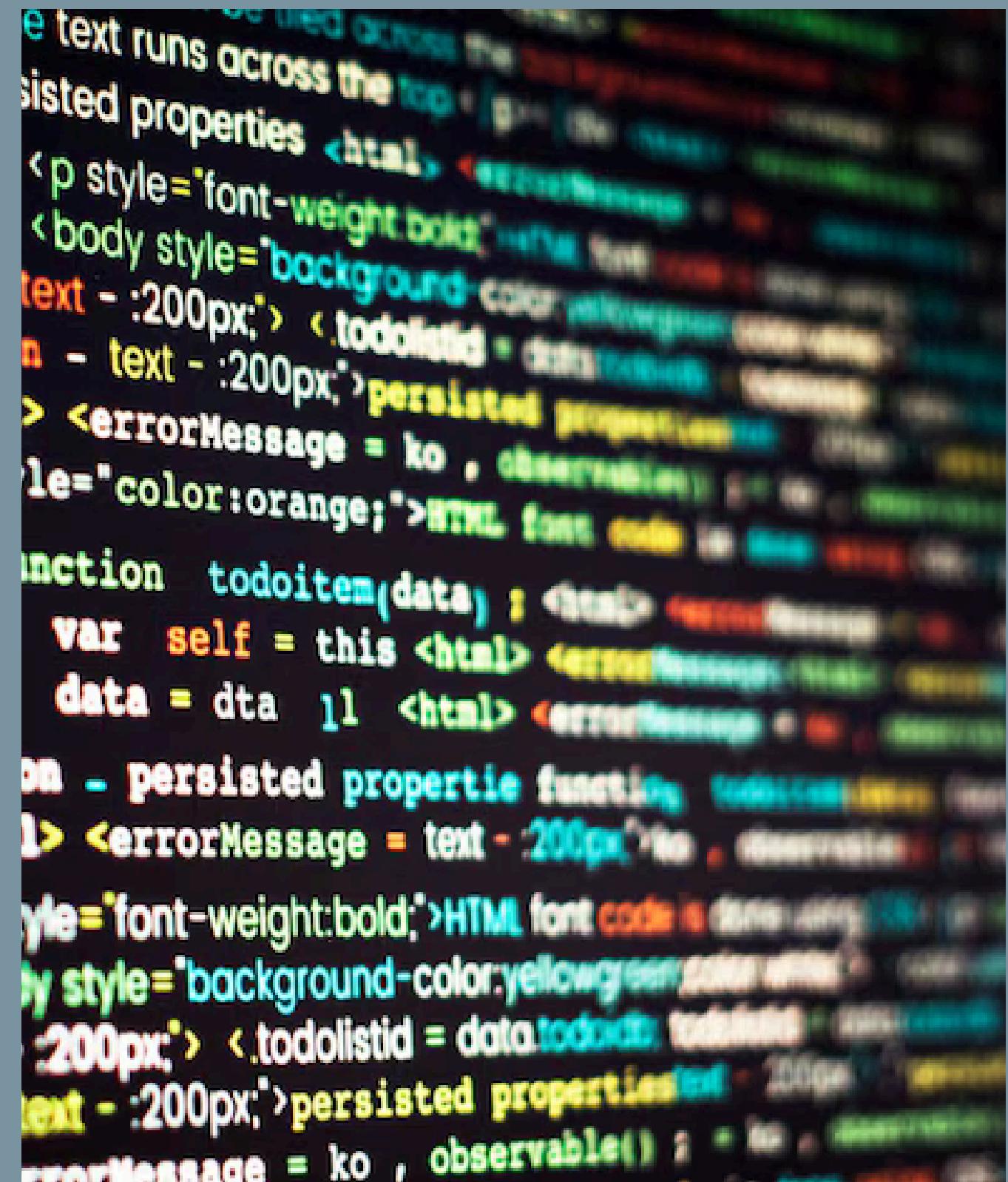
    return 0;
}
```

CodeGenVisitor

- CodeGenVisitor parcourt l'AST et génère des instructions.
- Création d'un CFG (Control Flow Graph) pour chaque fonction.
- Chaque CFG contient :
 - une table des symboles,
 - des blocs de base (BasicBlock),
 - les instructions intermédiaires.

Fonctionnalités étendues de notre compilateur

IR



```
e text runs across the top < p> the  
sisted properties < html> < errorMes-  
< p style="font-weight:bold">HTML font code is done using  
< body style="background-color:yellowgreen"> color  
text - :200px;> <.todoListId = data.todoListId> id - 100px  
n - text - :200px;> persisted properties  
> <errorMessage = ko , observable()> error  
le="color:orange;">HTML font code is done using  
unction todoitem(data) ; <body> <errorMes-  
var self = this < html> <errorMes-  
data = data || < html> <errorMes-  
pn - persisted properties function < errorMes-  
l> <errorMessage = text - :200px;> ko , observable()  
yle="font-weight:bold;">HTML font code is done using  
by style="background-color:yellowgreen"> color  
:200px;> <.todoListId = data.todoListId> id - 100px  
text - :200px;> persisted properties  
<errorMessage = ko , observable()> error
```

IR =

IRInstr



Basic Block



CFG

a chaque opération est associée une fonction gen qui la traduit en assembleur en tenant compte des registres

```
/
IRInstr::IRInstr(BasicBlock *bb_, Operation op, Type t,
                  const vector<Parameter> &parameters)
    : block(bb_), op(op), outType(t), parameters(parameters) {}

/*
* Génère le code assembleur x86-64 correspondant à l'instruction IR
* @param os Le flux de sortie pour écrire le code assembleur
* @param cfg Le graphe de flux de contrôle associé
*/
void IRInstr::genAsm(ostream &os, CFG *cfg)
{
    switch (op)
    {
        case add:
            generateBinaryOperation("addl", os, cfg);
            break;
        case sub:
            generateBinaryOperation("subl", os, cfg);
            break;
        case mul:
            generateBinaryOperation("imull", os, cfg);
            break;
        case cmpNZ:
            generateCompareNotZero(os, cfg);
            break;
        case div:
```

```
/*
* Génère le code assembleur pour une opération binaire (add, sub, etc)
* @param op Le mnémonique assembleur (ex: "addl", "subl")
* Gère les différents cas d'allocation de registres
*/
void IRInstr::generateBinaryOperation(const string &op, ostream &os,
                                       CFG *cfg)
{

    int firstRegister =
        cfg->getRegisterIndexForSymbol(get<shared_ptr<Symbol>>(parameters[0]));
    int secondRegister =
        cfg->getRegisterIndexForSymbol(get<shared_ptr<Symbol>>(parameters[1]));
    int destRegister =
        cfg->getRegisterIndexForSymbol(get<shared_ptr<Symbol>>(parameters[2]));
    if (firstRegister == cfg->scratchRegister)
    {
        os << "movl " << get<shared_ptr<Symbol>>(parameters[0])->offset
           << "(%rbp), %" << registers32[firstRegister] << endl;
    }
    if (firstRegister == cfg->scratchRegister &&
        secondRegister == cfg->scratchRegister &&
        destRegister == cfg->scratchRegister)
```

L'**IRInstr** comprend un enum de tout les opérations pouvant être effectuées

$$\text{IR} = \text{Basic Block} + \text{CFG}$$

Ce sont des Blocs d'instructions IF
enchaînées sans branchement
interne

```
// ===== Classe BasicBlock =====
// Représente un bloc de base dans le CFG
class BasicBlock
{
public:
    BasicBlock(CFG *cfg, string entry_label);
    void gen_asm(ostream &o); // Génère l'assembleur du bloc

    shared_ptr<Symbol> add_IRInstr(IRInstr::Operation op, Type t,
                                    vector<Parameter> parameters);

    BasicBlock *exit_true;          // Bloc suivant si condition vraie
    BasicBlock *exit_false;         // Bloc suivant si condition fausse (sinon jump inconditionnel)
    bool visited;                  // Indique si ce bloc a déjà été généré (utile pour éviter les doublons)
    string label;                  // Label du bloc (nom unique)
    CFG *cfg;                      // CFG auquel appartient ce bloc
    vector<IRInstr> instrs;        // Liste des instructions IR dans ce bloc
    string test_var_name;          // Nom de la variable de test (pour if / while, etc.)
};
```

Ils portent un label unique et pointent vers d'éventuels blocs suivants

IR =
IRinstr
+
Basic
Block
+
CFG

Le CFG contrôle le flux et orchestre l'exécution des Basic Blocks

Le CFG gère aussi la table des symboles et optimise l'allocation des variables

Le CFG ajoute l'épilogue et le prologue de chaque fonction

IR

```
int main() {  
    int counter = 0;           BBstart  
    while(counter < 5) {       BBtest  
        counter++;            BBbody  
    }  
    return counter;           BBend  
}
```

`int counter = 0;`

se traduit par une déclaration de variable counter de type int, suivie d'une affectation à la constante 0.

par l'IR:

`Idconst 0, counter`

En assembleur:

`movl $0, %eax`

`movl %eax, -offset(%rbp)`

Analyse de vivacité et allocation de registres

```
/**  
 * Effectue l'allocation de registres pour le CFG  
 * Utilise l'analyse de vivacité et le graphe d'interférence  
 */  
void CFG::computeRegisterAllocation()  
{  
    InstructionLivenessAnalysis liveInfo = computeLiveInfo();  
    map<shared_ptr<Symbol>, vector<shared_ptr<Symbol>>>  
        interferenceGraph = buildInterferenceGraph(liveInfo);  
    RegisterAllocationSpillData spillInfo = findColorOrder(interferenceGraph, 7);  
  
    registerAssignment = assignRegisters(spillInfo, interferenceGraph, 7);  
}
```

- computeLiveInfo détermine quelles variables sont actives (ou ‘vivantes’) à chaque instant d’exécution.
- buildInterferenceGraph repère les variables dont les plages de vie se chevauchent et qui ne peuvent donc pas partager le même registre.
- L’étape findColorOrder vient ensuite et, avec assignRegisters, effectue la répartition effective sur nos sept registres disponibles

Avec

computeRegisterAllocation nous optimisons l’allocation mémoire avec des shared pointer

Difficultés rencontrées

- X86 ARM
 - Organisation des BB pour les opérations conditionnelles





Fonction visitVar_decl_stmt()

- Gère les déclarations avec ou sans initialisation.
- Si une expression est présente, elle est évaluée et affectée.
- Sinon, initialisation par défaut à 0.
- Garantit un comportement mémoire fiable.

```
antlr4::Any
CodeGenVisitor::visitVar_decl_stmt(ifccParser::Var_decl_stmtContext *ctx)
{
    Type type;
    if (ctx->TYPE()->toString() == "int")
    {
        type = Type::INT;
    }
    else if (ctx->TYPE()->toString() == "char")
    {
        type = Type::CHAR;
    }
    else if (ctx->TYPE()->toString() == "void")
    {
        VisitorErrorListener::addError(ctx, "Can't create a variable of type void");
    }
    for (auto &memberCtx : ctx->var_decl_member())
    {
        string varName = memberCtx->ID()->toString();
        addSymbolToSymbolTableFromContext(memberCtx, varName, type);
        if (memberCtx->expr())
        {
            shared_ptr<Symbol> symbol = getSymbolFromSymbolTableByContext(memberCtx, varName);
            shared_ptr<Symbol> source = visit(memberCtx->expr()).as<shared_ptr<Symbol>>();
            curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {symbol, source});
        }
        else
        {
            // Initialisation implicite à 0
            shared_ptr<Symbol> symbol = getSymbolFromSymbolTableByContext(memberCtx, varName);
            auto zero = curCfg->current_bb->add_IRInstr(IRInstr::ldconst, Type::INT, {"0"});
            curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {symbol, zero});
        }
    }
    return 0;
}
```





Fonction Fonction visitIf_else()

- Création de blocs pour if, else et un bloc de fin.
- Évaluation de la condition via cmpNZ.
- Transition explicite vers le bloc correspondant.
- Structure claire du flux de contrôle.

```
antlr4::Any CodeGenVisitor::visitIf_else(ifccParser::If_elseContext *ctx)
{
    shared_ptr<Symbol> result =
        visit(ctx->expr()).as<shared_ptr<Symbol>>();
    string elseBBLabel = ".L" + to_string(nextLabel);
    nextLabel++;
    string endBBLabel = ".L" + to_string(nextLabel);
    nextLabel++;

    BasicBlock *baseBlock = curCfg->current_bb;
    BasicBlock *trueBlock = new BasicBlock(curCfg.get(), "");
    BasicBlock *elseBlock = new BasicBlock(curCfg.get(), elseBBLabel);
    BasicBlock *endBlock = new BasicBlock(curCfg.get(), endBBLabel);

    trueBlock->exit_true = endBlock;
    elseBlock->exit_true = endBlock;
    endBlock->exit_true = baseBlock->exit_true;

    baseBlock->add_IRInstr(IRInstr::cmpNZ, Type::INT, {result});

    curCfg->add_bb(trueBlock);
    visit(ctx->if_block);

    curCfg->add_bb(elseBlock);
    visit(ctx->else_block);

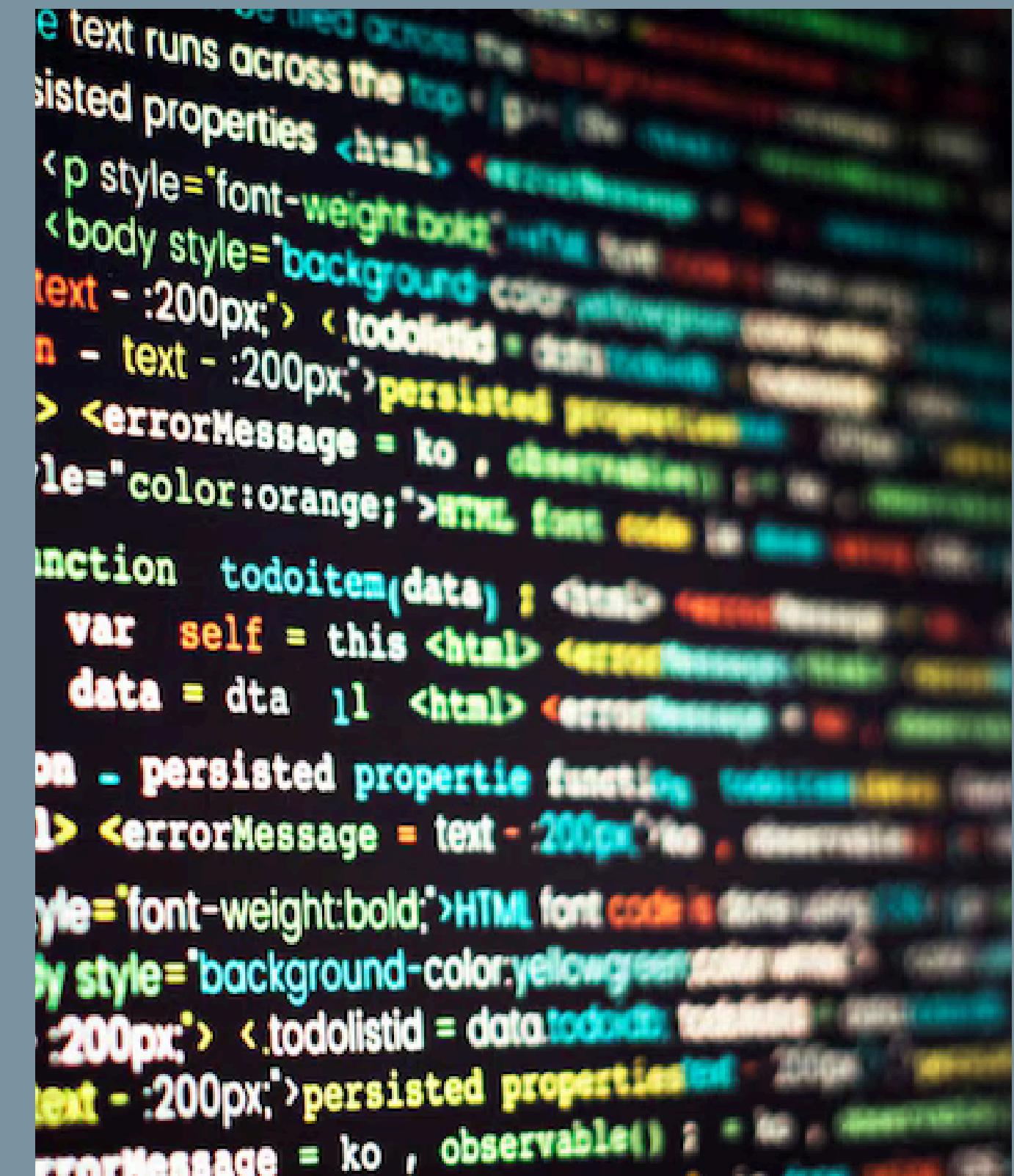
    curCfg->add_bb(endBlock);

    baseBlock->exit_true = trueBlock;
    baseBlock->exit_false = elseBlock;
    return 0;
}
```



Fonctionnalités étendues de notre compilateur

- Initialisation à la déclaration
 - Type CHAR
 - Opérateurs `+=`, `-=`, `*=`
 - Opérateurs `++`, `--` (Incrémantation et décrémantation)
 - Opérateurs paresseux `&&` et `||`



Initialisation à la déclaration

• • • •

```
var_decl_stmt : TYPE var_decl_member (',' var_decl_member)* ';' ;
var_decl_member: ID ('=' expr)?;
```

- Possibilité d'écrire **int x = 5;** ou **char c = 'A';**
- Extension de la grammaire pour gérer les affectations dans la déclaration
- Traitement sémantique : type vérifié et valeur calculée
- Génération IR : Idconst ou expr, suivi de var_assign

```
if (memberCtx->expr())
{
    shared_ptr<Symbol> symbol = getSymbolFromSymbolTableByContext(memberCtx, varName);
    shared_ptr<Symbol> source = visit(memberCtx->expr()).as<shared_ptr<Symbol>>();
    curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {symbol, source});
}
else
{
    // Initialisation implicite à 0
    shared_ptr<Symbol> symbol = getSymbolFromSymbolTableByContext(memberCtx, varName);
    auto zero = curCfg->current_bb->add_IRInstr(IRInstr::ldconst, Type::INT, {"0"});
    curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {symbol, zero});
}
```

On gère les deux cas :

- initialisation explicite
- initialisation implicite à 0

• • • • •



Support du type char

```
TYPE : INT | CHAR | VOID ;
```

```
enum class Type { INT, CHAR, VOID };
```

- Nouveau type ajouté à la grammaire : **char**
- Gestion du type dans l'analyse sémantique et l'IR
- Allocation mémoire adaptée : 1 octet
- Conversion ASCII des caractères ('A', 'z', etc.)
- Support dans les expressions, les paramètres et les retours

```
else if (ctx->CHAR_LITERAL() != nullptr)
{
    string val =
        to_string(static_cast<int>(ctx->CHAR_LITERAL()->toString()[1]));
    source =
        curCfg->current_bb->add_IRInstr(IRInstr::ldconst, Type::CHAR, {val});
}
```

Extrait fonction visitVal





Opérateurs composés : +=, -=, *=

```
var_assign_stmt: ID ('=' | '+=' | '-=' | '*=' | '/=') expr ';' ;
```

- Nouveaux opérateurs ajoutés à la grammaire :
+=, -=, *=
- Reconnus comme des formes d'affectation spécifiques
- Interprétés comme des instructions load → op → assign
- Traduit en IR avec add, sub, mul + var_assign
- Génération d'instructions x86 optimisées (addl, subl, imull)

```
if (op == "=")
{
    curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {symbol, source});
}
else
{
    IRInstr::Operation instr;
    if (op == "+=")
        instr = IRInstr::add;
    else if (op == "-=")
        instr = IRInstr::sub;
    else if (op == "*=")
        instr = IRInstr::mul;
    else if (op == "/=")
        instr = IRInstr::div;

    shared_ptr<Symbol> temp = curCfg->create_new_tempvar(Type::INT);
    curCfg->current_bb->add_IRInstr(IRInstr::ldvar, Type::INT, {symbol});
    curCfg->current_bb->add_IRInstr(instr, Type::INT, {symbol, source, temp});
    curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {symbol, temp});
}
return 0;
}
```

Extrait fonction visitVar_assign_stmt





Incrémantation et décrémantation : ++, --

- Prise en charge des **opérateurs ++ et --** en préfixe et suffixe
- Intégration dans la grammaire ANTLR
- Distinction dans la sémantique (avant ou après utilisation)
- Traduction en IR : inc / dec et gestion de la valeur renournée
- Génération d'instructions x86 **addl \$1 ou subl \$1**

```
antlrcpp::Any CodeGenVisitor::visitPreIncDec(ifccParser::PreIncDecContext *ctx)
{
    shared_ptr<Symbol> symbol = getSymbolFromSymbolTableByContext(ctx, ctx->ID()->getText());

    // Récupérer l'opérateur à partir du texte brut
    std::string fullText = ctx->getText();
    std::string op = fullText.substr(0, 2); // Les deux premiers caractères

    IRIInstr::Operation operation = (op == "++") ? IRIInstr::inc : IRIInstr::dec;

    // Incrémentation avant utilisation
    curCfg->current_bb->add_IRInstr(operation, Type::INT, {symbol});

    return symbol;
}
```

```
expr : ID ('++' | '--') #postIncDec
      | ('++' | '--') ID #preIncDec
```

```
antlrcpp::Any CodeGenVisitor::visitPostIncDec(ifccParser::PostIncDecContext *ctx)
{
    shared_ptr<Symbol> symbol = getSymbolFromSymbolTableByContext(ctx, ctx->ID()->getText());

    // Récupérer l'opérateur à partir du texte brut
    std::string fullText = ctx->getText();
    std::string op = fullText.substr(fullText.size() - 2); // Les deux derniers caractères

    IRIInstr::Operation operation = (op == "++") ? IRIInstr::inc : IRIInstr::dec;

    // Sauvegarde de la valeur avant incrémentation
    shared_ptr<Symbol> temp = curCfg->create_new_tempvar(Type::INT);
    curCfg->current_bb->add_IRInstr(IRInstr::ldvar, Type::INT, {symbol});
    curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {temp, symbol});

    // Incrémentation
    curCfg->current_bb->add_IRInstr(operation, Type::INT, {symbol});

    return temp;
}
```





Les opérateurs logiques paresseux `//`, `&&`

- Évaluation en court-circuit :
 - `&&` n'évalue le 2^e opérande que si le 1^{er} est vrai.
 - `//` n'évalue le 2^e opérande que si le 1^{er} est faux.
- Sémantique optimisée :
 - Évite les calculs inutiles (ex : division par zéro, déréférencement NULL).
- Traduction en IR :
 - Utilisation de sauts conditionnels (`je/jne`) pour contrôler le flux.
- Génération ASM :
 - Étiquettes (`.L_true`, `.L_false`) pour gérer les branchements.
- Différence clé :
 - Paresseux (`&&///`) vs toujours évalués (`&/|`).

```
| expr '||' expr #logicalOr  
| expr '&&' expr #logicalAnd
```

```
antlrcpp::Any CodeGenVisitor::visitLogicalOr(ifccParser::LogicalOrContext *ctx) {  
    string falseLabel = ".L" + to_string(nextLabel++);  
    string endLabel = ".L" + to_string(nextLabel++);  
  
    shared_ptr<Symbol> left = visit(ctx->expr(0)).as<shared_ptr<Symbol>>();  
    shared_ptr<Symbol> result = curCfg->create_new_tempvar(Type::INT);  
  
    // Évaluation paresseuse - si gauche est vrai, résultat est vrai  
    curCfg->current_bb->add_IRInstr(IRInstr::cmpNZ, Type::INT, {left});  
    curCfg->current_bb->add_IRInstr(IRInstr::ldconst, Type::INT, {"1"});  
    curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {result, left});  
  
    BasicBlock* falseBB = new BasicBlock(curCfg.get(), falseLabel);  
    BasicBlock* endBB = new BasicBlock(curCfg.get(), endLabel);  
  
    curCfg->current_bb->exit_false = falseBB;  
    curCfg->current_bb->exit_true = endBB;  
  
    curCfg->add_bb(falseBB);  
    shared_ptr<Symbol> right = visit(ctx->expr(1)).as<shared_ptr<Symbol>>();  
    curCfg->current_bb->add_IRInstr(IRInstr::var_assign, Type::INT, {result, right});  
  
    curCfg->add_bb(endBB);  
    return result;
```



Quelques jeux de tests intéressants





Test fonction + putchar

Ce qu'il teste :

- Appel de la fonction standard putchar
- Manipulation d'un char sous forme arithmétique ('A' + 1)
- Retour de valeur d'une fonction appelée

Pourquoi il est utile :

- Vérifie le support des fonctions natives
- Montre que les expressions sur char sont bien gérées
- Teste le passage de littéraux + calculs à une fonction externe

```
#include <stdio.h>

int main() {
    int x=17;
    int y=42;
    int ret = putchar('A'+1);
    return ret;
}
```





Test opérateur composé

Ce qu'il teste :

- L'opérateur d'affectation composé *=

Pourquoi il est utile :

- Valide l'implémentation de vos opérateurs facultatifs (+=, -=, *=)
- Vérifie que le compilateur décompose bien a *= b en :
 - chargement
 - multiplication
 - réaffectation

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

```
int main() {
    int a = 5;
    int b = 10;

    a *= b;

    return 0;
}
```





Test suffixe $x++$

Ce qu'il teste :

- L'opérateur de post-incrémantation $a++$

Pourquoi il est utile :

- Vérifie que le comportement suffixe est bien respecté :
 - la valeur est retournée avant l'incrément
- Montre une combinaison non triviale avec d'autres opérations

```
int main() {  
    int a = 3;  
    int b = a++;  
    return b + a * 10;  
}
```





Test préfixe $++x$

💡 **Ce qu'il teste :**

- L'opérateur de pré-incrémantation $++a$

🎯 **Pourquoi il est utile :**

- Vérifie que l'incrémantation est faite avant le retour
- Utile à comparer avec 13_post_incr.c pour illustrer la différence

```
int main() {  
    int a = 7;  
    int b = ++a;  
    return b + a * 10;  
}
```



.....

Test boucle while avec compteur

💡 Ce qu'il teste :

- Boucle while avec condition simple
- Incrémentation dans le corps de boucle

🎯 Pourquoi il est utile :

- Vérifie que la structure de contrôle while fonctionne correctement
- Montre que le CFG boucle bien sur lui-même
- Vérifie aussi ++ dans un contexte itératif

```
#include <stdio.h>

int main() {
    int counter = 0;
    while (counter < 5) {
        counter++;
    }
    return counter;
}
```



.....

Test Erreur sémantique

Ce qu'il teste :

- Utilisation d'une variable non initialisée

Pourquoi il est utile :

- Permet de tester la gestion des erreurs sémantiques
- Vérifie que le compilateur ne compile pas un programme incorrect

```
int main()
{
    int a = 2;
    int b;
    return a + b;
}
```



DÉMONSTRATION

