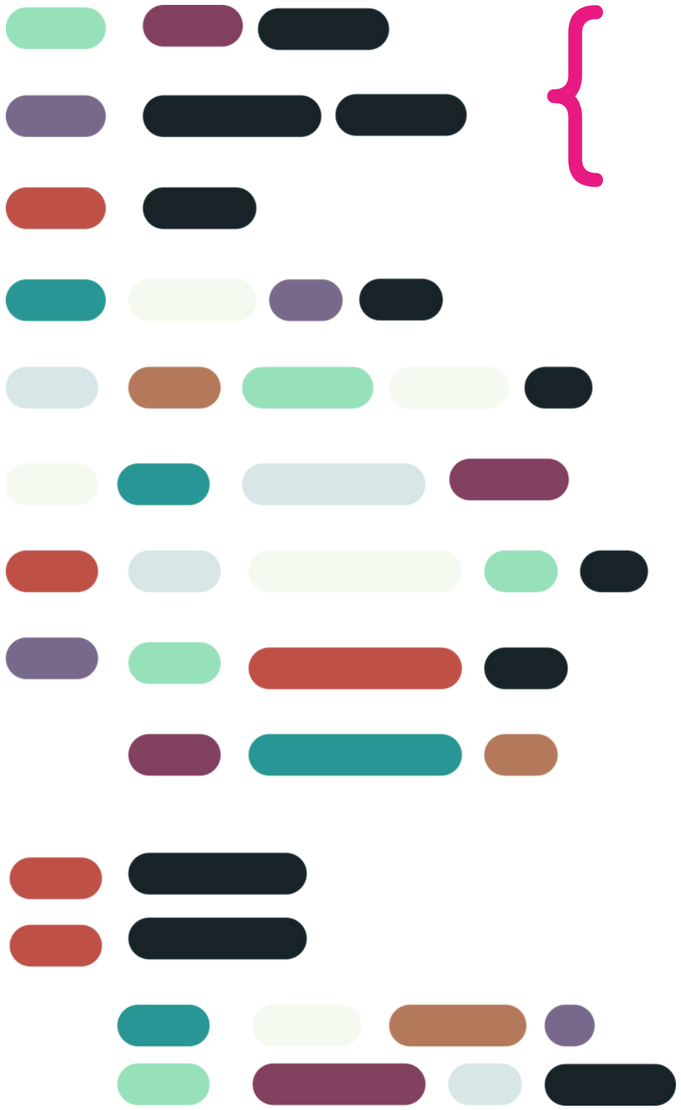


Compilateur C vers x86/ARM

< Projet PLD-Comp • H4233 >

Avant de commencer !

<Contexte, Objectif et Structure du PLD-COMP>

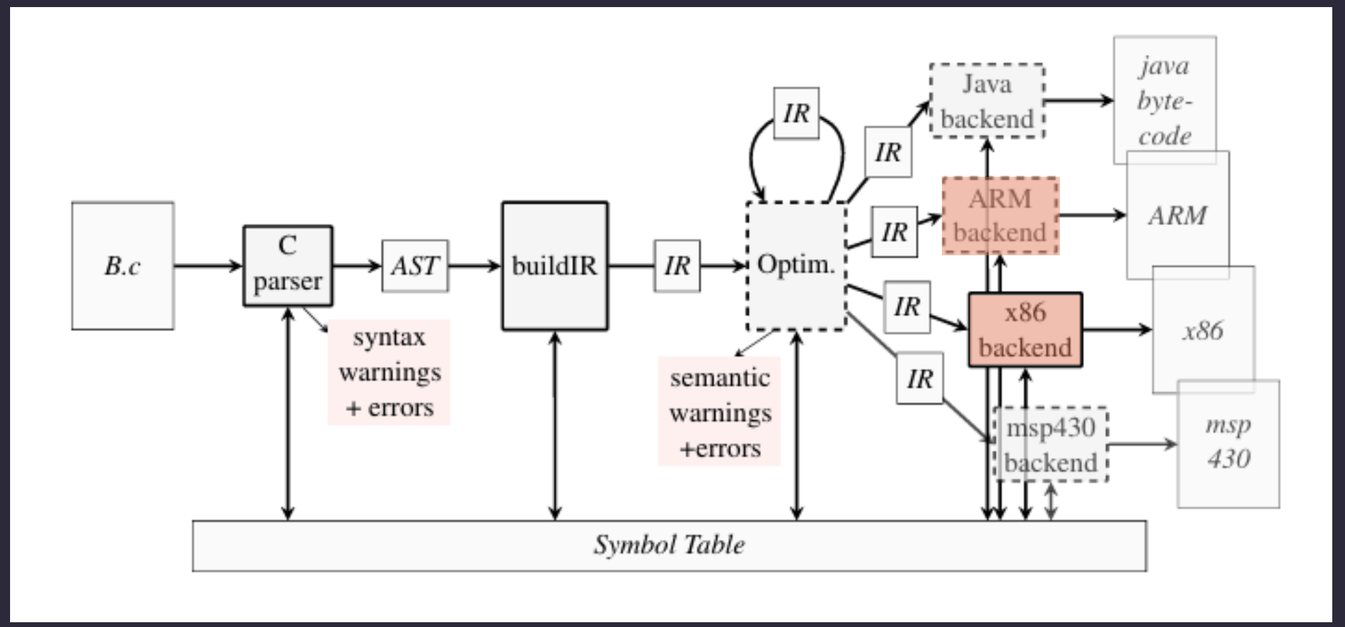


Figure 1 - Le compilateur de nos rêves devenus réalité

Plan de la soutenance

01 Architecture du compilateur

Chaine de compilation, Grammaire & Parsing, Visiteurs

02 Fonctionnalités du langage C supportées

Déclarations, Opérations, Appels & Définition de Fonctions et encore plus

03 Tests et démonstrations

Câs simples, Cas extrêmes, Cas KO + Démo Live !

04 Optimisations & Extensions

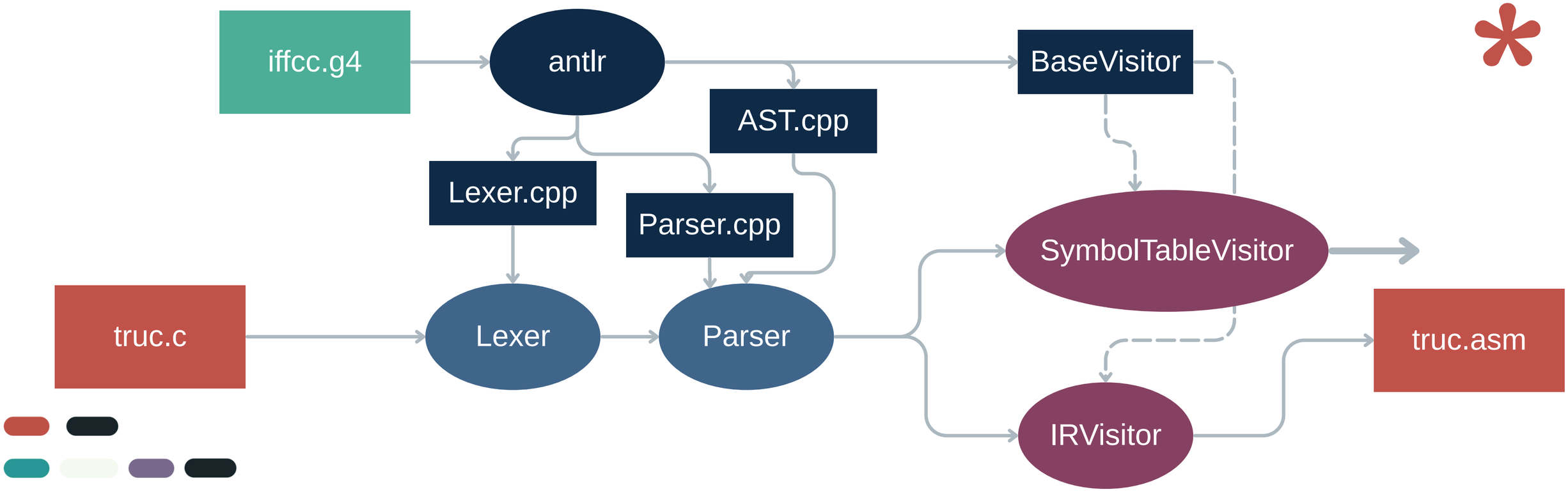
Backend ARM & Gestion technique spécifique

05 Gestion de Projet

& perspectives à venir

01 { .. Architecture du compilateur } . .

Chaine de compilation



Grammaire & Parsing ANTLR4

Visiteurs et Analyse

SymbolTableVisitor

Structure / Classe	Fichier	Rôle dans le projet
SymbolTableVisitor	SymbolTableVisitor.h/.cpp	Visiteur de l’AST ANTLR, construit les symbol tables par fonction et bloc
SymbolTable	SymbolTable.h	Table de hachage locale (1 par bloc) : associe noms de variables à leur type et offset
Function	Function.h	Structure regroupant le type de retour, les symbol tables par fonction, et le stack offset
Symbol	Function.h	Structure de variable : nom, offset, type (ex : int x → -4)
Type	Type.h	Enumération des types (INT, CHAR, DOUBLE, VOID)

IRGenerator

Structure / Classe	Fichier	Rôle dans le projet
IRGenerator	IRGenerator.h/.cpp	Visiteur de l’AST, génère des instructions IR pour chaque expression ou instruction
CFG	CFG.h/.cpp	Graphe de flot de contrôle, regroupe les BasicBlock, fonctions, offsets, symbol tables
BasicBlock	BasicBlock.h/.cpp	Bloc de base avec une séquence d’instructions IR + transitions conditionnelles
IRInstr	IRInstr.h/.cpp	Instruction typée (ex : add, ldconst, call, ret...)

Visiteurs et Analyse

02 { .. Fonctionnalités supportées } . .

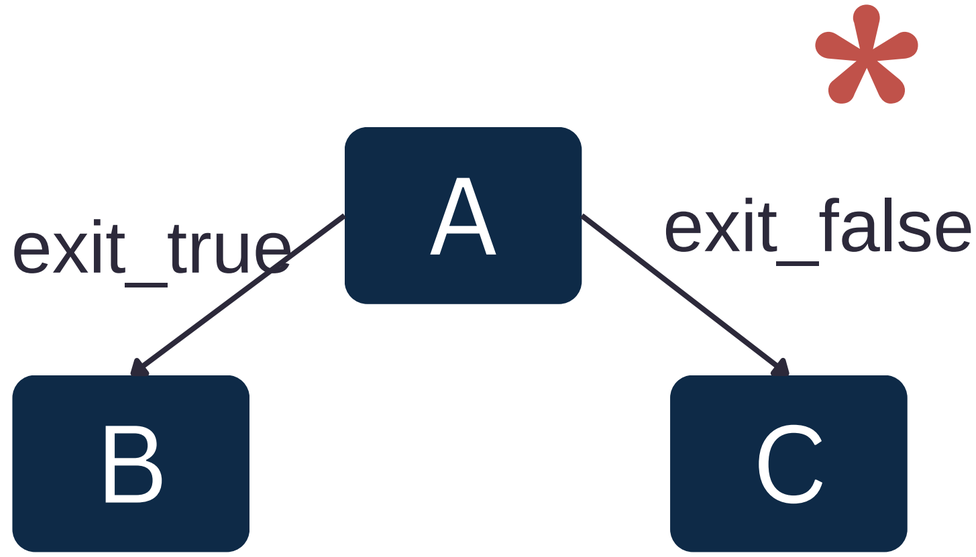
Declaration, Assignement, Return

Opérations & Comparaison

Portée & Shadowing

Structures Conditionelles: If Then Else

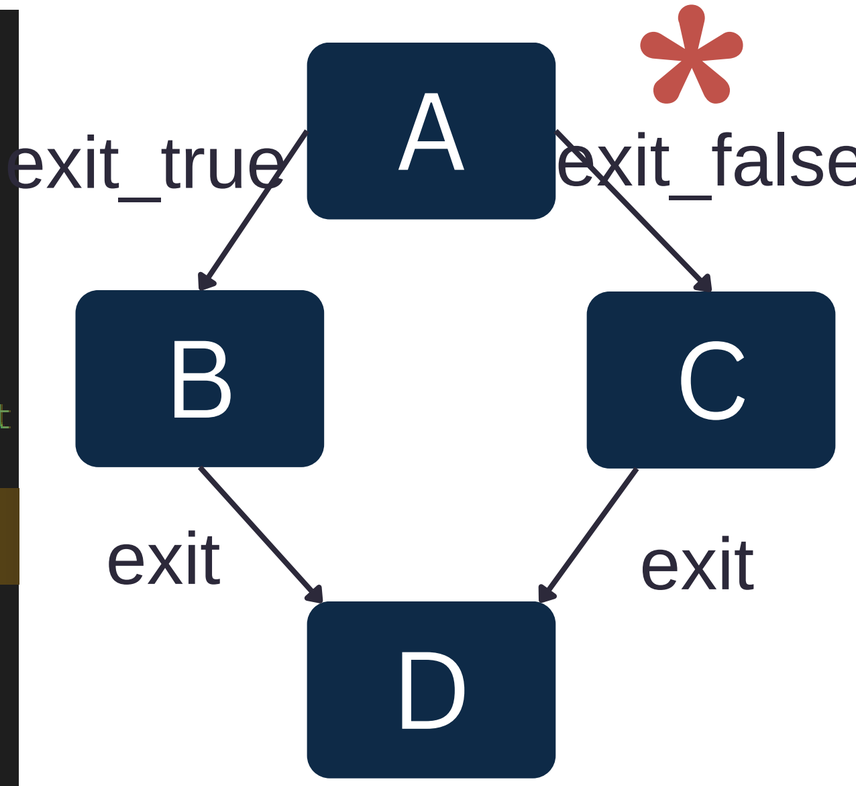
```
antlrcpp::Any IRGenerator::visitIf_stmt(ifccParser::If_stmtContext *ctx)
{
    visit(ctx->expr()); // Évaluation de l'expression conditionnelle
    int bbs_size = cfg->bbs.size(); // Taille actuelle de la liste des blocs de base

    std::string if_label = "label" + to_string(bbs_size);
    BasicBlock *true_exit_bb = new BasicBlock(cfg, if_label);
    BasicBlock *previous_bb = cfg->current_bb;
    cfg->current_bb->exit_true = true_exit_bb; // Sortie vraie du bloc de base actuel
    cfg->current_bb = true_exit_bb;
    cfg->add_bb(true_exit_bb);

    if (ctx->block(1)) // Si une autre instruction "else" est présente

    {
        bbs_size = cfg->bbs.size();
        string else_label = "label" + to_string(bbs_size);
        BasicBlock *false_exit_bb = new BasicBlock(cfg, else_label);
        previous_bb->exit_false = false_exit_bb; // Sortie fausse du bloc de base actuel
        cfg->add_bb(false_exit_bb);
    }
    else
    {
        previous_bb->exit_false = nullptr; // Pas de sortie fausse
    }
```



Structures Conditionelles: If Then Else

```
bbs_size = cfg->bbs.size();
string next_label = "label" + to_string(bbs_size);
BasicBlock *next_bb = new BasicBlock(cfg, next_label); // Bloc de base suivant
cfg->add_bb(next_bb);

previous_bb->exit_true->exit = next_bb; // Sortie du bloc then vers le bloc suivant
if (previous_bb->exit_false)
{
    previous_bb->exit_false->exit = next_bb; // Sortie du bloc else vers le bloc suivant
}
```

```
visit(ctx->block(0)); // Visite du bloc "then"
if (previous_bb->exit_false)
{
    cfg->current_bb = previous_bb->exit_false;
    visit(ctx->block(1));  // Visite du bloc "else" s'il existe
}
```

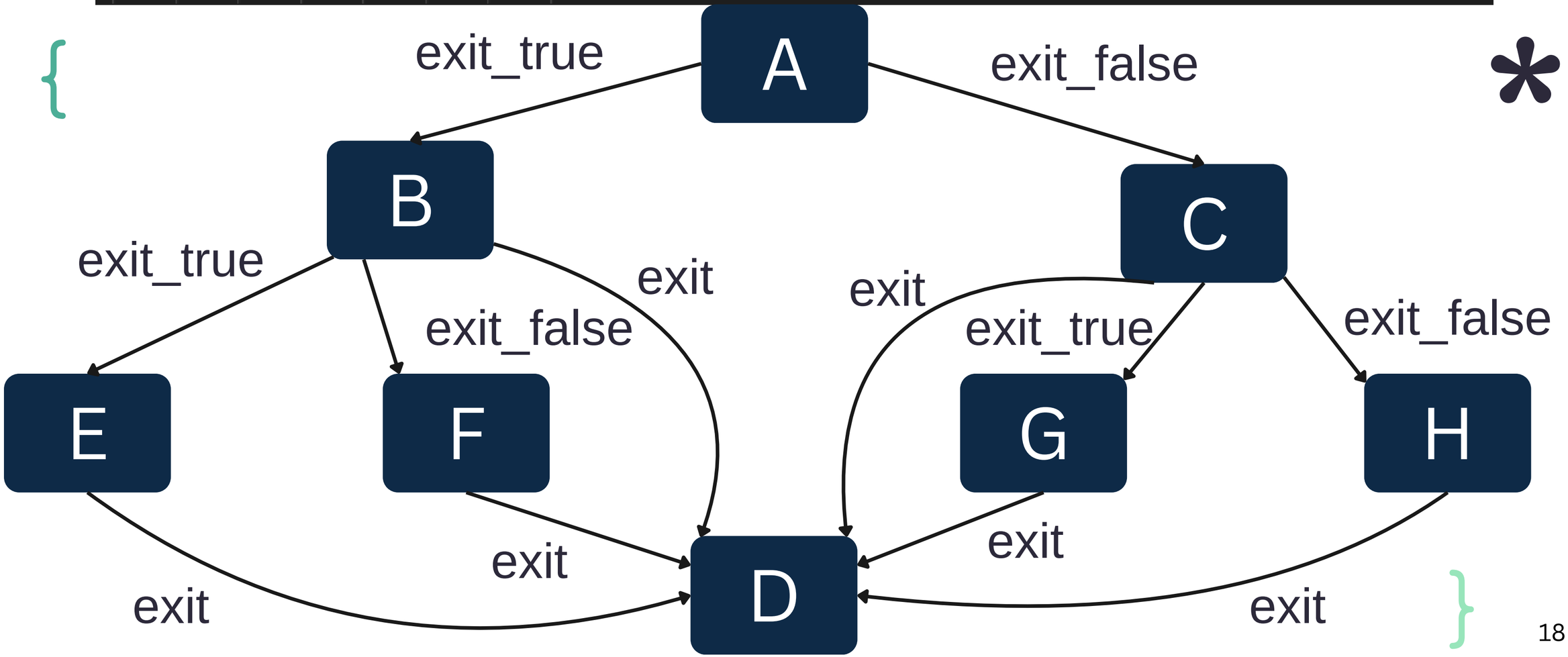


Structures Conditionelles: If Then Else

```
{
    ..
    Et les blocs
    conditionnels
    imbriqués ?
                }
```

Structures Conditionelles: If Then Else

```
next_bb->exit = previous_bb->exit;/* s'il s'agit d'un bloc conditionnel imbriqué
                                     sa sortie sera celle du dernier bloc conditionnel*/
```



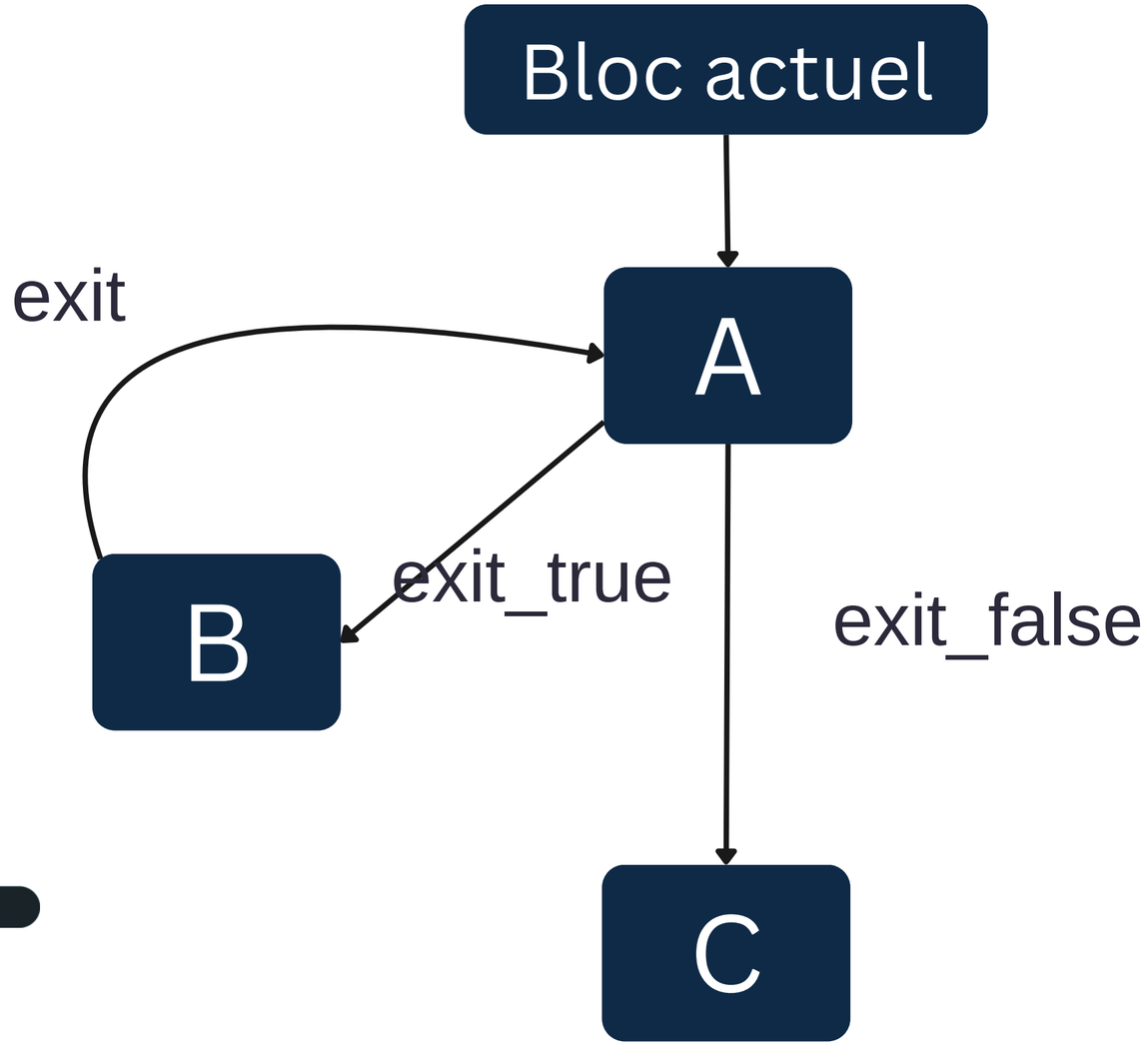
Structures Conditionelles: If Then Else

```
cfg->current_bb = next_bb; // Passage au bloc suivant

std::string else_or_next_label; // Étiquette de sortie pour le bloc "else" ou le bloc suivant
if (previous_bb->exit_false)
{
    else_or_next_label = previous_bb->exit_false->label;
}
else
{
    else_or_next_label = next_bb->label;
}

previous_bb->add_IRInstr(IRInstr::cond_jump, INT, {if_label, else_or_next_label}); // Ajout de l'instruction de saut conditionnel
```

Structures Conditionelles: While

```
antlrcpp::Any IRGenerator::visitWhile_stmt(ifccParser::While_stmtContext *ctx)
{

    int bbs_size = cfg->bbs.size();
    std::string condition_bb_label = "label" + to_string(bbs_size); // Étiquette pour le bloc de condition
    BasicBlock *condition_bb = new BasicBlock(cfg, condition_bb_label);// Bloc de base pour la condition
    cfg->add_bb(condition_bb);
    bbs_size = cfg->bbs.size();
    std::string bb_true__label = "label" + to_string(bbs_size);
    BasicBlock *bb_true = new BasicBlock(cfg, bb_true__label);// Bloc de base pour la sortie vraie
    cfg->add_bb(bb_true);
    bbs_size = cfg->bbs.size();
    std::string bb_false_label = "label" + to_string(bbs_size);
    BasicBlock *bb_false = new BasicBlock(cfg, bb_false_label);// Bloc de base pour la sortie fausse
    cfg->add_bb(bb_false);
```

Structures Conditionelles: While (CFG)

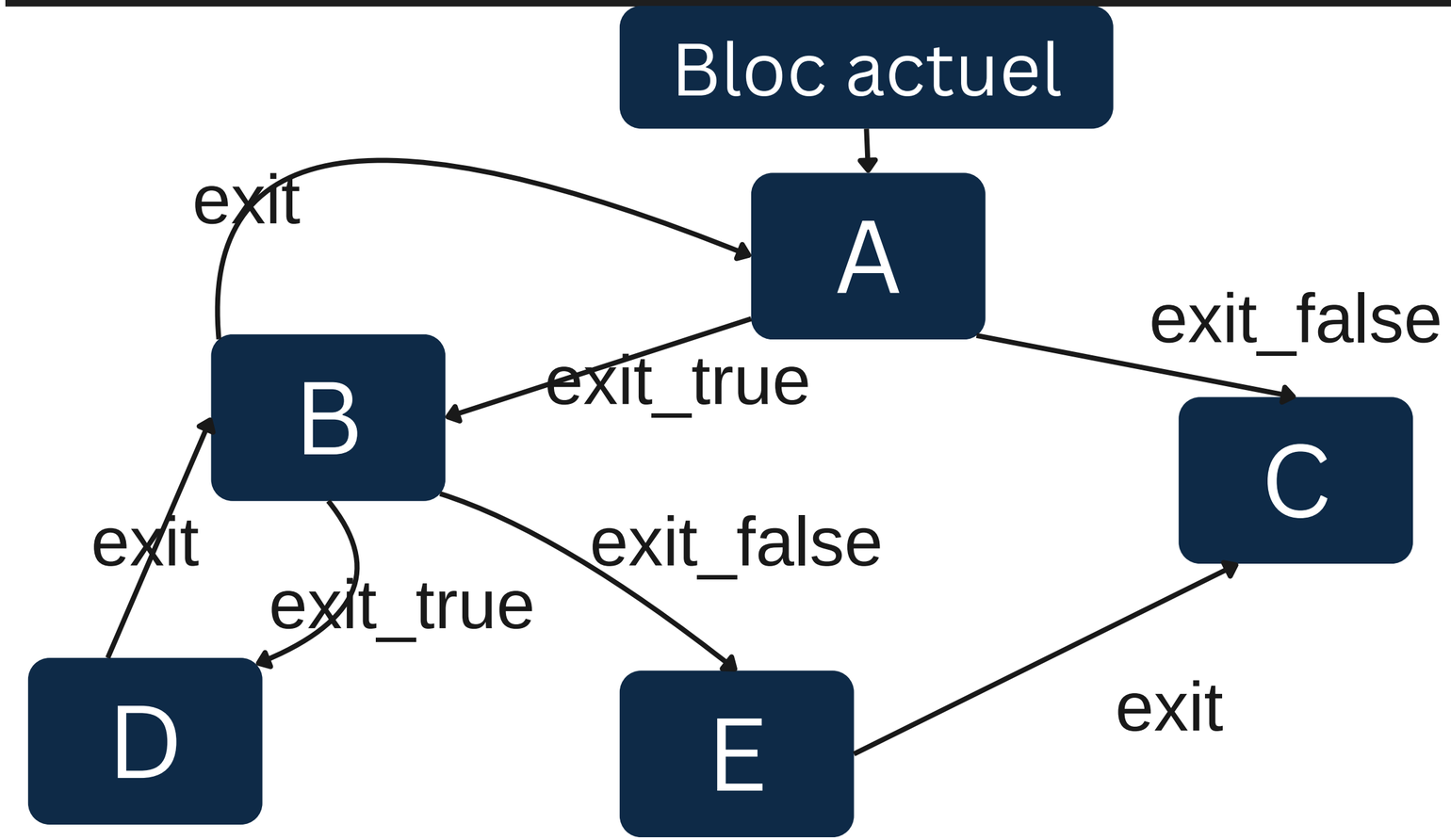


Structures Conditionelles: While

```
condition_bb->exit_true = bb_true; // Sortie vraie du bloc de condition
condition_bb->exit_false = bb_false;// Sortie fausse du bloc de condition

bb_true->exit = condition_bb;
cfg->current_bb->exit = condition_bb;
cfg->current_bb->add_IRInstr(IRInstr::jump, INT, {condition_bb_label});// Saut inconditionnel vers le bloc de condition
cfg->current_bb = condition_bb;
visit(ctx->expr());// Évaluation de l'expression conditionnelle
cfg->current_bb->add_IRInstr(IRInstr::cond_jump, INT, {bb_true__label, bb_false_label}); // Saut conditionnel vers la sortie vraie ou fausse

cfg->current_bb = bb_true;// Bloc de base pour la sortie vraie
visit(ctx->block());// Visite du bloc de la boucle
cfg->current_bb = bb_false; // Bloc de base pour la sortie fausse (gestion des boucles en fin de visitBlock)
```

Structures Conditionelles: While imbriqués

```
bb_false->exit = cfg->current_bb->exit; //Gestion de blocs conditionnels imbriqués
```




```
//bloc actuel
while(a>0) //bloc A
{  //bloc B
    b = b+1;
    a  = a-1;
    while(b>0){ //bloc D
        c= c+1;
        b= b-1;
    }
    //bloc E
    a = b;
}
//bloc C
```

Appel de fonctions

```
int i = 0;
auto it = cfg->functions[funcName].symbolTable[0]->table.begin();
Type type = cfg->functions[funcName].returnType;
while (ctx->expr(i)) // Vérification de l'existence de paramètres
{

    visit(ctx->expr(i)); // Évaluation de l'expression du paramètre
    //Assigner la valeur stockée dans %eax au bon registre de paramètres
    cfg->current_bb->add_IRInstr(IRInstr::assign_param, lastExprType, {to_string(i)});

    ++i;
    ++it;
}
// Ajout de l'instruction d'appel de fonction
cfg->current_bb->add_IRInstr(IRInstr::call, type, {funcName});
```

Définition de fonctions

```
antlrcpp::Any IRGenerator::visitFunctionDef(ifccParser::FunctionDefContext *ctx)
{
    std::string funcName = ctx->ID()->getText();

    BasicBlock *function_bb = new BasicBlock(cfg, funcName); // Bloc de base pour la fonction
    cfg->currentFunction = funcName; // Changement de la fonction courante
    cfg->add_bb(function_bb);
    cfg->current_bb = function_bb;

    SymbolTable *symbolTable = cfg->functions[funcName].symbolTable.at(0);

    if (symbolTable != nullptr) // Vérification de l'existence de paramètres
    {
        auto it = symbolTable->table.begin();
        int i = 0;
        for (; it != symbolTable->table.end(); ++it, ++i)
        {
            cfg->current_bb->add_IRInstr(IRInstr::load_param_from_reg, INT, {to_string(it->second.symbolOffset), to_string(i)});
            //Récupération de la valeur du registre et stockage dans le bon emplacement
        }
    }

    cfg->currentST_index = 0;
    cfg->last_ST_index = 0;
```

Définition de fonctions

```
// visit du code de la fonction
visit(ctx->block());

// Ajout du prologue en tête
function_bb->add_IRInstrAtTop(IRInstr::prologue, INT, {to_string(cfg->stack_allocation)});
```

04 {

Optimisation et extensions

}

05 { .. Organisation du Projet } . .

Sprint n° 1 :

Objectifs principaux	Responsables
- Types int, char (32 bits, ASCII)	Zineb, Saad
- Constantes (5, 'A' → 65)	Zineb, Saad
- Opérations arithmétiques +, -, *, /, %	Zineb, Saad
- Bitwise : `	Saad, Maroun, Peter
- Comparateurs ==, !=, <, <=, >, >=	Saad
- Unaires -, !	Saad
- Déclarations partout dans un bloc {}	Hafsa
- Affectations simples et chaînées a = b = 3	Nihal, Junior
- return avec expression	Maroun, Peter
- ✅ Chaque fonctionnalité est testée (cas simples, imbriqués, cas limites)	Peter, Maroun

Sprint n° 2 :

Objectifs principaux	Responsables
Structures if, else, while	HAFSA
Emboîtement de blocs ({}) → scoping dynamique	HAFSA
Table des symboles chaînée avec gestion de portées imbriquées	HAFSA
Shadowing supporté (ex : redéclaration de int x dans un bloc interne)	HAFSA
Appels standard : getchar(), putchar()	NIHAL & JUNIOR
double et conversion	NIHAL & JUNIOR
return à n'importe quel niveau	ZINEB
Fonctions, implémentation initiale	PETER & MAROUN
Suite des tests	PETER & MAROUN

Sprint n° 3 :

Objectifs principaux	Responsables
• Analyse statique : ◦ Variable non déclarée → erreur ◦ Double déclaration → erreur ◦ Variable déclarée non utilisée → warning	ZINEB
Backend ARM avec offset dynamique (éviter les erreurs d’offsets dynamiques)	SAAD
Suite des fonctions	MAROUN, HAFSA, ZINEB
Tableaux	NIHAL
Tests complexes et démonstration	PETER
Documentation des différents problèmes	JUNIOR
Préparation de la soutenance	SAAD, ZINEB

Perspectives à venir … dans un futur proche

Fonctionnalité	Pourquoi c’est facile ou complexe	Limitation / État actuel
• Gestion des fonctions utilisateur sur ARM	IR déjà unifié et backend-agnostique. Adapter assign_param et load_param_from_reg pour ARM suffit.	Géré uniquement sur x86.
Support complet de getchar() / putchar() + ARM	Même logique que pour x86. Ajouter le mapping ARM dans IRInstr::gen_asm_arm.	Partiellement supporté, uniquement sur x86
Gestion du type double sur ARM	Type déjà reconnu (Type::DOUBLE). Il faut ajouter les instructions ARM (mov.f64, add.f64, etc.).	Implémenté uniquement sur x86.

Perspectives à venir … dans un futur proche

switch / case	Traduction en suite de cond_jump dans l’IR.
Opérateurs / &&	Se traduit facilement dans l'IR existant.
Opérateurs +=, -=, ++, --, <<, >>	Se traduit facilement dans l'IR existant.
For / do ... whil	S’inspirer de While

Perspectives à venir … dans un futur moins proche

Fonctionnalité	Pourquoi c’est facile ou complexe
Propagation de constantes	Nécessite une passe d'optimisation après génération d'IR.
Propagation de variables constantes	Nécessite une analyse de data-flow.
Tableaux (1D)	Extension grammaire + modèle mémoire (adresses + accès indirect).
Pointeurs	Gestion fine des accès mémoire indirects.
break / continue	Simple au niveau IR → ajout de BasicBlock de sortie / skip.
Chaînes de caractères (char[])	Gestion des littéraux et de leur stockage mémoire.

Perspectives à venir … dans un futur lointain

Fonctionnalité	Pourquoi c’est facile ou complexe
Variables globales	Gestion d’une section .data globale + offsets indépendants des frames locaux
Types float/inttypes.h	Backend flottant complet à développer, surtout complexe en ARM
Opérateurs << >>	Extension grammaire + modèle mémoire (adresses + accès indirect).
Structures / Unions	Refactor du symbol table et du modèle mémoire
.h et .c séparés	??
Préprocesseur	nécessit un changement complet d’architecture