COMPILATEUR CPYRR VM

Compilateur complet avec machine virtuelle pour le langage de programmation CPYRR.

GHODBANE Rachid

Licence 3 Informatique – Université Jean Monnet 2025/2026

Table des matières

1 Introduction

Code source disponible sur GitHub :
https://github.com/rvsh0x/CPYRR_Compiler_VM

La compilation de langages de programmation représente un domaine fondamental de l'informatique, combinant théorie des langages formels, algorithmes de traitement de structures arborescentes et techniques de gestion de la mémoire. Ce rapport présente la conception et l'implémentation complète d'un compilateur pour le langage de programmation CPYRR, incluant une machine virtuelle pour l'exécution des programmes compilés.

Le compilateur CPYRR transforme un fichier source en représentation intermédiaire (fichiers texte), puis la machine virtuelle charge et exécute ces fichiers sur une pile d'exécution. Cette architecture en deux phases permet une séparation claire entre compilation et exécution, facilitant le débogage et l'évolution du système.

Le projet, développé en langage C, comprend plus de 50 fichiers sources organisés en modules fonctionnels, couvrant toutes les phases de compilation depuis l'analyse lexicale jusqu'à l'exécution sur machine virtuelle.

1.1 Objectifs

Les objectifs principaux de ce projet sont :

— Implémenter un compilateur complet couvrant toutes les phases classiques : analyse lexicale, syntaxique, sémantique et génération de code intermédiaire

— Concevoir une machine virtuelle avec gestion de la portée et adressage mémoire

— Développer un système de gestion d'erreurs contextuel et informatif

— Valider l'approche par représentation intermédiaire textuelle

1.2 Structure du rapport

Ce rapport est organisé comme suit : la section 2 présente le contexte théorique et les concepts fondamentaux de compilation. La section 3 détaille l'architecture et la conception du système. La section 4 expose l'implémentation détaillée de chaque composant avec des exemples de code tirés du projet. La section 5 présente les résultats et l'évaluation. Enfin, la section 6 conclut et propose des perspectives d'évolution.

2 Contexte théorique et fondements

2.1 Phases de compilation

La compilation d'un langage de programmation s'effectue traditionnellement en plusieurs phases distinctes, chacune transformant la représentation du programme source en une forme plus proche de l'exécution.

2.1.1 Analyse lexicale

L'analyse lexicale (ou tokenisation) transforme le flux de caractères source en une séquence de tokens. Formellement, soit Σ l'alphabet du langage source, l'analyseur lexical implémente une fonction :

$$\mathcal{L} : \Sigma^* \to \mathcal{T}^* \tag{1}$$

où $\mathcal{T}$ est l'ensemble des tokens possibles. Un token est une unité lexicale atomique : mot-clé, identificateur, constante, opérateur, etc. Cette phase utilise des automates finis déterministes (AFD) générés à partir d'expressions régulières.

Automate fini déterministe L'automate peut être modélisé comme un quintuplet $A = (Q, \Sigma, \delta, q_0, F)$ où :

— Q est l'ensemble fini d'états
— Σ est l'alphabet d'entrée (ensemble des caractères possibles)
— $\delta : Q \times \Sigma \to Q$ est la fonction de transition totale
— $q_0 \in Q$ est l'état initial
— $F \subseteq Q$ est l'ensemble des états finaux (acceptants)

La fonction de transition étendue $\delta^* : Q \times \Sigma^* \to Q$ est définie récursivement :

$$\delta^*(q, \varepsilon) = q \tag{2}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a) \quad \text{pour } w \in \Sigma^*, a \in \Sigma \tag{3}$$

où ε est le mot vide. Un mot $w \in \Sigma^*$ est accepté par l'automate si et seulement si $\delta^*(q_0, w) \in F$.

Expressions régulières Les expressions régulières définissent les patterns lexicaux. Soit R une expression régulière, le langage $L(R)$ qu'elle dénote est défini inductivement :

$$L(\emptyset) = \emptyset \tag{4}$$
$$L(\varepsilon) = \{\varepsilon\} \tag{5}$$
$$L(a) = \{a\} \quad \text{pour } a \in \Sigma \tag{6}$$
$$L(R_1|R_2) = L(R_1) \cup L(R_2) \tag{7}$$
$$L(R_1 R_2) = L(R_1) \cdot L(R_2) \tag{8}$$
$$L(R^*) = L(R)^* \tag{9}$$

où $L(R)^* = \bigcup_{i=0}^{\infty} L(R)^i$ est la fermeture de Kleene.

Construction de l'automate Le théorème de Kleene établit l'équivalence entre expressions régulières et automates finis. L'algorithme de Thompson permet de construire un automate non-déterministe (AFN) à partir d'une expression régulière, puis la construction de sous-ensembles (subset construction) permet d'obtenir un AFD équivalent.

Pour une expression régulière R, l'automate A_R accepte exactement le langage $L(R)$:

$$L(A_R) = L(R) \tag{10}$$

2.1.2 Analyse syntaxique

L'analyse syntaxique (ou parsing) construit une représentation structurée du programme à partir de la séquence de tokens. Soit $G = (V, T, P, S)$ une grammaire context-free où V est l'ensemble des symboles non-terminaux, T est l'ensemble des symboles terminaux (tokens), P est l'ensemble des productions, et $S \in V$ est l'axiome.

Grammaire context-free Une grammaire context-free $G = (V, T, P, S)$ définit un langage $L(G)$ où :

— V et T sont disjoints ($V \cap T = \emptyset$)

— $P \subseteq V \times (V \cup T)^*$ est un ensemble fini de productions de la forme $A \to \alpha$ où $A \in V$ et $\alpha \in (V \cup T)^*$

— $S \in V$ est le symbole de départ

La relation de dérivation $\Rightarrow_G$ est définie : si $A \to \beta \in P$, alors $\gamma A \delta \Rightarrow_G \gamma \beta \delta$ pour tout $\gamma, \delta \in (V \cup T)^*$.

La fermeture réflexive et transitive $\Rightarrow_G^*$ définit le langage :

$$L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\} \tag{11}$$

Arbre de dérivation Un arbre de dérivation (ou arbre syntaxique concret) pour une dérivation $S \Rightarrow_G^* w$ est un arbre étiqueté où :

— La racine est étiquetée S

— Les feuilles sont étiquetées par des symboles terminaux formant w

— Les nœuds internes sont étiquetés par des non-terminaux

— Pour un nœud A avec enfants $X_1, X_2, \ldots, X_n$, il existe une production $A \to X_1 X_2 \ldots X_n$

L'analyseur syntaxique implémente une fonction :

$$\mathcal{P} : \mathcal{T}^* \to \mathcal{A} \tag{12}$$

où $\mathcal{A}$ est l'ensemble des arbres syntaxiques abstraits (AST). Les AST sont des versions simplifiées des arbres de dérivation, ne conservant que l'information essentielle pour la génération de code.

Parsing LALR(1) L'algorithme LALR(1) (Look-Ahead Left-to-Right, Rightmost derivation) est une variante optimisée de LR(1). Il construit une table d'analyse à partir de la grammaire.

Soit G une grammaire, l'algorithme LALR(1) construit :

— Un ensemble d'états $I = \{I_0, I_1, \ldots, I_n\}$ où chaque état I_i est un ensemble d'items LR(0)

— Une fonction de transition GO(I, X) qui calcule l'état atteint depuis I en lisant X

— Des tables ACTION et GOTO pour guider le parsing

Un item LR(0) est une production avec un point : $A \to \alpha \bullet \beta$ indique que α a été reconnu et β est attendu.

L'algorithme de parsing utilise une pile d'états et une pile de symboles :

1. Initialisation : empiler l'état initial I_0
2. Pour chaque token a :
   — Consulter ACTION[s, a] où s est l'état au sommet
   — Si ACTION[s, a] = shift t : empiler a et t
   — Si ACTION[s, a] = reduce $A \to \beta$: dépiler $|\beta|$ symboles, puis empiler A et GOTO[s', A] où s' est le nouvel état au sommet
   — Si ACTION[s, a] = accept : accepter
   — Si ACTION[s, a] = error : rejeter

2.1.3 Analyse sémantique

L'analyse sémantique vérifie la cohérence du programme selon les règles du langage : déclarations, types, portée des identificateurs. Cette phase enrichit l'AST avec des annotations sémantiques (types, numéros de déclaration).

Système de types Un système de types formel peut être défini comme un tuple $(\mathcal{T}, \sqsubseteq, \text{type})$ où :

— $\mathcal{T}$ est l'ensemble des types

— $\sqsubseteq$ est une relation de sous-typage (ordre partiel)

— type : Expr $\to \mathcal{T}$ est une fonction d'inférence de type

Les règles de typage sont définies par un système de règles d'inférence. Par exemple, pour l'addition :

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sqsubseteq \text{num} \quad \tau_2 \sqsubseteq \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \tag{13}$$

où Γ est un environnement de types (mapping des identificateurs vers leurs types) et $\vdash$ dénote la relation de typage.

Vérification de types La vérification de types consiste à s'assurer que chaque expression a un type valide selon les règles du langage. Formellement, pour un AST t, on vérifie :

$$\exists \tau \in \mathcal{T} : \Gamma \vdash t : \tau \tag{14}$$

Si cette condition n'est pas satisfaite, une erreur de type est signalée.

Environnement de types L'environnement de types Γ est une fonction partielle :

$$\Gamma : \mathsf{Id} \rightharpoonup \mathcal{T} \tag{15}$$

L'environnement est étendu lors de la déclaration d'une variable :

$$\Gamma[x : \tau] = \Gamma \cup \{(x, \tau)\} \tag{16}$$

2.1.4 Génération de code

La génération de code produit une représentation exécutable. Dans notre cas, nous générons une représentation intermédiaire (RI) sous forme de fichiers texte, qui sera ensuite interprétée par une machine virtuelle.

Représentation intermédiaire Une représentation intermédiaire RI est une fonction qui transforme un AST annoté en une séquence d'instructions :

$$RI : \mathcal{A}_{\text{annoté}} \to \mathcal{I}^* \tag{17}$$

où $\mathcal{I}$ est l'ensemble des instructions de la machine virtuelle. La RI permet de séparer les préoccupations : le compilateur génère la RI, la machine virtuelle l'exécute.

2.2 Représentation intermédiaire

Une représentation intermédiaire permet de séparer la compilation de l'exécution. Formellement, une RI est une abstraction qui capture la sémantique du programme source de manière indépendante de la machine cible.

Définition formelle Soit $\mathcal{P}$ l'ensemble des programmes source et $\mathcal{M}$ l'ensemble des machines cibles, une représentation intermédiaire est une fonction :

$$RI : \mathcal{P} \to \mathcal{RI} \tag{18}$$

où $\mathcal{RI}$ est l'espace des représentations intermédiaires. La propriété fondamentale est que la RI préserve la sémantique :

$$\text{sémantique}(p) = \text{sémantique}(RI(p)) \tag{19}$$

Avantages théoriques Les avantages incluent :

- **Portabilité** : $\forall m \in \mathcal{M}, \exists \text{codegen}_m : \mathcal{RI} \to \text{Code}_m$ tel que $\text{sémantique}(RI(p)) = \text{sémantique}(\text{codegen}_m(RI(p)))$
- **Débogage** : Format lisible permettant l'inspection : $\text{inspectable}(RI(p)) = \text{true}$
- **Optimisation** : Possibilité d'appliquer des transformations $T : \mathcal{RI} \to \mathcal{RI}$ préservant la sémantique : $\text{sémantique}(RI(p)) = \text{sémantique}(T(RI(p)))$
- **Modularité** : Séparation des préoccupations : compilation et exécution sont des fonctions indépendantes

Formats de RI Dans notre implémentation, la RI est représentée sous forme textuelle structurée :

$$RI(p) = (\mathcal{L}, \mathcal{D}, \mathcal{R}, \mathcal{R}\rceil\}, \mathcal{A}) \tag{20}$$

où :

— $\mathcal{L}$ est la table lexicographique

— $\mathcal{D}$ est la table des déclarations

— $\mathcal{R}$ est la table des représentations de types

— $\mathcal{R}\rceil\}$ est la table des régions

— $\mathcal{A}$ est l'ensemble des arbres syntaxiques par région

2.3 Gestion de la portée et adressage

La gestion de la portée (scope) détermine la visibilité des identificateurs selon leur contexte de déclaration. Soit $\mathcal{R}$ l'ensemble des régions, organisé en arbre hiérarchique où parent(r) désigne la région parente de r.

Structure de portée Formellement, une structure de portée est un arbre $(\mathcal{R}, \text{parent})$ où :

— $\mathcal{R}$ est l'ensemble fini de régions

— parent : $\mathcal{R} \setminus \{r_0\} \to \mathcal{R}$ définit la hiérarchie

— $r_0 \in \mathcal{R}$ est la région racine (portée globale)

— La relation ancêtre(r, k) est définie récursivement :

$$\text{ancêtre}(r, 0) = r \tag{21}$$

$$\text{ancêtre}(r, k+1) = \text{parent}(\text{ancêtre}(r, k)) \quad \text{si ancêtre}(r, k) \neq r_0 \tag{22}$$

Environnement de déclarations Un environnement de déclarations Δ associe chaque région à un ensemble de déclarations :

$$\Delta : \mathcal{R} \to \mathcal{P}(\text{Decl}) \tag{23}$$

où Decl est l'ensemble des déclarations. Pour une région r, $\Delta(r)$ contient toutes les déclarations locales à r.

2.3.1 Résolution de noms

L'algorithme de résolution d'un identificateur id dans une région r peut être formalisé comme :

$$\text{resolve}(id, r) = \begin{cases} \text{decl}(id, r) & \text{si decl}(id, r) \neq \bot \\ \text{resolve}(id, \text{parent}(r)) & \text{si parent}(r) \neq \bot \\ \bot & \text{sinon} \end{cases} \tag{24}$$

où $\text{decl}(id, r) = \begin{cases} d & \text{si } \exists d \in \Delta(r) : \text{nom}(d) = id \\ \perp & \text{sinon} \end{cases}$ retourne la déclaration de id dans la région r ou $\perp$ si absente.

Cette définition garantit la règle de portée lexicale : une déclaration locale masque les déclarations des régions parentes.

Input: Identificateur id, région courante r
Output: Numéro de déclaration ou $\perp$

```
r_current ← r;
while r_current ≥ 0 do
    decl ← chercher_dans_region(id, r_current);
    if decl ≠ ⊥ then
        return decl;
    end
    r_current ← parent(r_current);
end
return ⊥;
```

Algorithm 1: Résolution d'un identificateur

2.3.2 Chaînage statique

Le chaînage statique permet d'accéder aux variables des régions parentes. Pour une région r de profondeur d, le chaînage statique maintient d pointeurs vers les bases de code des régions ancêtres :

$$\text{BC}_i(r) = \text{BC}(\text{ancêtre}(r, i)) \quad \text{pour } i \in [0, d-1] \tag{25}$$

Principe du chaînage statique Le chaînage statique est basé sur la structure lexicale (statique) du programme, contrairement au chaînage dynamique qui suit la chaîne d'appels. Pour une variable v déclarée dans une région r_d et utilisée dans une région r_u, on remonte dans la hiérarchie lexicale jusqu'à trouver r_d.

Formellement, le chaînage statique maintient pour chaque zone Z_r à l'index $\text{BC}(r)+i$ (pour $i \in [1, \text{NIS}(r)]$) la base de code de la région à i niveaux au-dessus dans la hiérarchie lexicale :

$$\text{pile}[\text{BC}(r) + i] = \text{BC}(\text{ancêtre}(r, i)) \quad \text{pour } i \in [1, \text{NIS}(r)] \tag{26}$$

Construction du chaînage lors de l'empilement Lors de l'empilement d'une nouvelle zone pour une région $r_{\text{appelée}}$ depuis une région $r_{\text{appelante}}$, trois cas se présentent selon l'évolution du NIS :

1. **NIS augmente** ($\text{NIS}(r_{\text{appelée}}) > \text{NIS}(r_{\text{appelante}})$) :
   - La région appelée est plus imbriquée que la région appelante
   - On copie tous les chaînages de la région appelante
   - On ajoute un nouveau chaînage vers la région appelante
   - Formellement : $\text{pile}[\text{BC}_{\text{nouveau}} + 1] = \text{BC}_{\text{ancien}}$ et pour $i \in [1, \text{NIS}(r_{\text{appelante}})]$: $\text{pile}[\text{BC}_{\text{nouveau}} + 1 + i] = \text{pile}[\text{BC}_{\text{ancien}} + i]$
2. **NIS stagne** ($\text{NIS}(r_{\text{appelée}}) = \text{NIS}(r_{\text{appelante}})$) :
   - Les deux régions sont au même niveau d'imbrication
   - On copie tous les chaînages de la région appelante
   - Formellement : pour $i \in [1, \text{NIS}(r_{\text{appelée}})]$: $\text{pile}[\text{BC}_{\text{nouveau}}+i] = \text{pile}[\text{BC}_{\text{ancien}}+i]$
3. **NIS décroît** ($\text{NIS}(r_{\text{appelée}}) < \text{NIS}(r_{\text{appelante}})$) :
   - La région appelée est moins imbriquée (retour vers un niveau supérieur)
   - On doit remonter dans la chaîne statique de la région appelante
   - On copie les chaînages depuis la région cible
   - Formellement : soit $k = \text{NIS}(r_{\text{appelante}}) - \text{NIS}(r_{\text{appelée}})$, alors $\text{BC}_{\text{cible}} = \text{pile}[\text{BC}_{\text{ancien}}+k]$ et pour $i \in [1, \text{NIS}(r_{\text{appelée}})]$: $\text{pile}[\text{BC}_{\text{nouveau}} + i] = \text{pile}[\text{BC}_{\text{cible}} + i]$

Algorithme d'empilement avec chaînage statique

Input: Numéro de région $r_{\text{appelée}}$
Output: Zone empilée avec chaînages statiques
$\text{BC}_{\text{ancien}} \leftarrow \text{BC}$;
$\text{taille}_{\text{courante}} \leftarrow \text{taille}(\text{region\_courante})$;
$\text{BC}_{\text{nouveau}} \leftarrow \text{BC}_{\text{ancien}} + \text{taille}_{\text{courante}}$;
$\text{NIS}_{\text{appelant}} \leftarrow \text{NIS}(\text{region\_courante})$;
$\text{NIS}_{\text{appele}} \leftarrow \text{NIS}(r_{\text{appelée}})$;
$\text{pile}[\text{BC}_{\text{nouveau}}] \leftarrow \text{BC}_{\text{ancien}}$ `// Sauvegarde BC ancien` ;
if $\mathit{NIS}_{\mathit{appele}} > 0$ **then**
  if $\mathit{NIS}_{\mathit{appele}} > \mathit{NIS}_{\mathit{appelant}}$ **then**
    `// Cas 1 : NIS augmente`
    $\text{pile}[\text{BC}_{\text{nouveau}} + 1] \leftarrow \text{BC}_{\text{ancien}}$;
    for $i \leftarrow 1$ **to** $\mathit{NIS}_{\mathit{appelant}}$ **do**
      $\text{pile}[\text{BC}_{\text{nouveau}} + 1 + i] \leftarrow \text{pile}[\text{BC}_{\text{ancien}} + i]$;
    end
  end
  else if $\mathit{NIS}_{\mathit{appele}} = \mathit{NIS}_{\mathit{appelant}}$ **then**
    `// Cas 2 : NIS stagne`
    for $i \leftarrow 1$ **to** $\mathit{NIS}_{\mathit{appele}}$ **do**
      $\text{pile}[\text{BC}_{\text{nouveau}} + i] \leftarrow \text{pile}[\text{BC}_{\text{ancien}} + i]$;
    end
  end
  else
    `// Cas 3 : NIS décroît`
    $k \leftarrow \text{NIS}_{\text{appelant}} - \text{NIS}_{\text{appele}}$;
    $\text{BC}_{\text{cible}} \leftarrow \text{pile}[\text{BC}_{\text{ancien}} + k]$;
    for $i \leftarrow 1$ **to** $\mathit{NIS}_{\mathit{appele}}$ **do**
      $\text{pile}[\text{BC}_{\text{nouveau}} + i] \leftarrow \text{pile}[\text{BC}_{\text{cible}} + i]$;
    end
  end
end
if *est_fonction*$(r_{\mathit{appelée}})$ **then**
  $\text{pile}[\text{BC}_{\text{nouveau}} + \text{NIS}_{\text{appele}} + 1] \leftarrow 0$ `// Zone retour` ;
end
$\text{BC} \leftarrow \text{BC}_{\text{nouveau}}$;
$\text{region\_courante} \leftarrow r_{\text{appelée}}$;

Algorithm 2: Empilement d'une zone avec gestion du chaînage statique

Adressage mémoire Soit Mem l'espace mémoire, organisé comme une séquence de cellules. Pour une variable v déclarée dans une région r_d et utilisée dans une région r_u, l'adresse est calculée via :

$$\text{adr}(v, r_d, r_u) = \text{BC}(\text{remonter}(r_u, \text{NIS}(r_u) - \text{NIS}(r_d))) + \delta(v, r_d) + \delta_{\text{retour}}(r_d) \tag{27}$$

où :

— $\text{BC}(r)$ est la base de code de la région r dans la pile d'exécution

— $\text{NIS}(r)$ est le numéro d'index de sauvegarde de la région r

— $\delta(v, r_d)$ est le déplacement de v dans sa région de déclaration

— $\delta_{\text{retour}}(r_d) = \begin{cases} 1 & \text{si } r_d \text{ est une fonction} \\ 0 & \text{sinon} \end{cases}$ est le décalage dû à la zone de retour

— $\text{remonter}(r, k)$ remonte k niveaux dans la chaîne statique

Algorithme de calcul d'adresse L'algorithme de calcul d'adresse utilise le chaînage statique pour remonter jusqu'à la région de déclaration :

Input: Numéro de déclaration d, région courante r_u
Output: Adresse mémoire de la variable
$r_d \leftarrow \text{region}(d)$;
$\delta \leftarrow \text{deplacement}(d)$;
$\text{NIS}_d \leftarrow \text{NIS}(r_d)$;
$\text{NIS}_u \leftarrow \text{NIS}(r_u)$;
$k \leftarrow \text{NIS}_u - \text{NIS}_d$;
if $k = 0$ **then**
  `// Variable locale`
  $\text{BC}_d \leftarrow \text{BC}$;
end
else if $k > 0$ **then**
  `// Variable dans une région parente`
  $\text{BC}_d \leftarrow \text{pile}[\text{BC} + k]$;
end
else
  `// Erreur : variable dans une région enfant`
  return *erreur*;
end
if $r_d = 0$ **then**
  `// Région globale`
  $\text{adr} \leftarrow \delta$;
end
else
  $\delta_{\text{retour}} \leftarrow \text{est\_fonction}(r_d)?1 : 0$;
  $\text{adr} \leftarrow \text{BC}_d + \delta + \delta_{\text{retour}}$;
end
return *adr*;

Algorithm 3: Calcul d'adresse d'une variable

Calcul de déplacement Pour une variable v dans une région r, le déplacement est calculé selon l'ordre des déclarations :

$$\delta(v, r) = \sum_{v' \in \text{declarations\_avant}(v,r)} \text{taille}(\text{type}(v')) \tag{28}$$

où declarations_avant(v, r) est l'ensemble des variables déclarées avant v dans r, et taille(τ) est la taille mémoire d'un type τ.

Pile d'exécution La pile d'exécution $\mathcal{S}$ est une séquence de zones, une par région active :

$$\mathcal{S} = [Z_0, Z_1, \ldots, Z_n] \tag{29}$$

où chaque zone Z_i correspond à une région r_i et contient :

— Les variables locales de r_i

— Les chaînages statiques vers les régions parentes

— La zone de retour (si r_i est une fonction)

Organisation d'une zone d'activation Chaque zone d'activation (activation record) pour une région r de profondeur d est organisée comme suit :

$$Z_r = [\text{BC}_{\text{ancien}}, \text{BC}_0, \text{BC}_1, \ldots, \text{BC}_{d-1}, \text{retour ?}, \text{param}_1, \ldots, \text{param}_p, \text{var}_1, \ldots, \text{var}_v] \tag{30}$$

où :

— $\text{BC}_{\text{ancien}}$: Base de code de la zone appelante (à l'index BC(r))

— BC_i : Chaînage statique vers la région à i niveaux au-dessus (aux indices BC$(r)+1$ à BC$(r) +$ NIS(r))

— retour : Zone de retour pour les fonctions (à l'index BC$(r) +$ NIS$(r) + 1$)

— param_i : Paramètres de la fonction/procédure

— var_i : Variables locales de la région

La taille totale d'une zone est :

$$\text{taille}(Z_r) = 1 + \text{NIS}(r) + \delta_{\text{retour}} + \sum_{i=1}^{p} \text{taille}(\text{type}(\text{param}_i)) + \sum_{i=1}^{v} \text{taille}(\text{type}(\text{var}_i)) \tag{31}$$

où $\delta_{\text{retour}} = \begin{cases} 1 & \text{si } r \text{ est une fonction} \\ 0 & \text{sinon} \end{cases}$.

Numéro d'Index de Sauvegarde (NIS) Le NIS d'une région r représente le nombre de régions ancêtres entre r et la région globale. Formellement :

$$\text{NIS}(r) = \begin{cases} 0 & \text{si } r = r_0 \text{ (région globale)} \\ \text{NIS}(\text{parent}(r)) + 1 & \text{sinon} \end{cases} \tag{32}$$

Le NIS détermine le nombre de chaînages statiques à maintenir dans chaque zone d'activation.

Les mécanismes classiques incluent :

— Pile de portées pour gérer l'imbrication

— Chaînage statique pour accéder aux variables des portées parentes

— Calcul d'adresses selon le type et la position dans la mémoire

3 Architecture et conception

3.1 Vue d'ensemble

Le système est organisé en deux composants principaux : le compilateur et la machine virtuelle. Cette séparation permet une indépendance complète entre les phases de compilation et d'exécution.

3.2 Architecture du compilateur

Le compilateur suit une architecture modulaire avec les composants suivants. Le point d'entrée principal orchestre toutes les phases de compilation : initialisation des tables, analyse syntaxique, analyse sémantique et génération de code intermédiaire.

Le code complet du point d'entrée principal est disponible en Annexe B.10 (page 66).

Les composants principaux sont :

3.2.1 Analyseur lexical

- Générateur : Lex
- Rôle : Transformation source $\rightarrow$ tokens
- Sortie : Séquence de tokens avec positions

3.2.2 Analyseur syntaxique

- Générateur : Yacc (LALR)
- Rôle : Construction de l'AST
- Sortie : Arbre syntaxique abstrait annoté

3.2.3 Analyseur sémantique

- Rôle : Vérifications de cohérence
- Entrée : AST + Tables symboles
- Sortie : AST enrichi avec informations sémantiques

3.2.4 Générateur de code intermédiaire

- Rôle : Production des fichiers TI
- Format : Fichiers texte structurés
- Contenu : Tables + Arbres

3.3 Architecture de la machine virtuelle

La machine virtuelle est organisée en modules :

3.3.1 Chargeurs

— Parsing des fichiers TI (Lex/Yacc dédiés)
— Reconstruction des structures en mémoire
— Validation de cohérence

3.3.2 Pile d'exécution

— Gestion de la mémoire par zones
— Chaînage statique pour portée
— Vérification d'initialisation

3.3.3 Interpréteur

— Évaluation d'expressions
— Exécution d'instructions
— Gestion des appels de fonctions

*Les diagrammes d'architecture détaillés sont disponibles en **Annexe A** (pages 59, 60 et 61).*

4 Implémentation détaillée

Cette section présente l'implémentation détaillée de chaque composant du système, en décrivant les structures de données, algorithmes et choix de conception.

4.1 Analyse lexicale

L'analyseur lexical est généré automatiquement par Lex à partir d'une spécification par expressions régulières. Cette approche garantit un scanner optimisé et maintenable.

4.1.1 Spécification lexicale

La grammaire lexicale du langage CPYRR définit :

— **Mots-clés** : `prog`, `var`, `type`, `struct`, `array`, `procedure`, `function`, etc.

— **Types de base** : `int`, `real`, `bool`, `char`, `string`

— **Constantes** : Entières (`[0-9]+`), réelles (`[0-9]+.[0-9]+`), caractères (`'...'`), chaînes (`"..."`), booléennes (`true`, `false`)

— **Identificateurs** : `[a-zA-Z][a-zA-Z0-9_]*`

— **Opérateurs** : Arithmétiques (`+`, `-`, `*`, `/`), booléens (`&&`, `||`, `!`), comparaisons (`=`, `<>`, `<`, `>`, etc.)

4.1.2 Flux de traitement lexical

Le processus d'analyse lexicale suit un flux séquentiel :

1. **Lecture du caractère** : Le scanner lit un caractère depuis le flux d'entrée
2. **Reconnaissance du pattern** : L'automate fini déterministe généré par Lex teste les expressions régulières dans l'ordre de déclaration
3. **Action associée** : Lorsqu'un pattern correspond, l'action associée est exécutée :
   — Mise à jour des positions (ligne, colonne, longueur)
   — Calcul du hash et recherche/insertion dans la table lexicographique
   — Retour du token correspondant avec sa valeur sémantique
4. **Gestion des commentaires** : Les commentaires sont consommés sans générer de token, mais les positions sont mises à jour pour préserver la localisation
5. **Gestion des espaces** : Les espaces et tabulations sont ignorés mais la colonne courante est incrémentée

4.1.3 Gestion des positions

Le tracking précis des positions (ligne, colonne, longueur) est essentiel pour la localisation des erreurs. Les variables globales `cc` (colonne courante), `cdt` (colonne début token) et `ltc` (longueur token courant) sont mises à jour pour chaque token reconnu.

Le mécanisme de tracking fonctionne ainsi :

— `cc` est incrémenté à chaque caractère lu (sauf retours à la ligne)

— `cdt` est fixé au début de chaque token reconnu

— `ltc` est fixé à la longueur du token reconnu

— Lors d'un retour à la ligne (`\n`), `cc` est réinitialisé à 1 et `ligne_courante` est incrémenté

4.1.4 Intégration avec la table lexicographique

Chaque identificateur est recherché puis inséré dans la table lexicographique, retournant un numéro unique utilisé dans tout le compilateur. Cette approche évite les doublons et garantit l'unicité des lexèmes.

Le processus d'intégration suit ces étapes :

1. Calcul du hash du lexème via la fonction $h(s) = \left(\sum_{i=0}^{n-1} \text{ord}(c_i)\right) \bmod 32$
2. Recherche dans la chaîne de collision associée au hash
3. Si trouvé, retour du numéro existant
4. Si non trouvé, insertion en tête de chaîne et retour du nouveau numéro

4.1.5 Fonction de hachage

La table lexicographique utilise une fonction de hachage pour un accès rapide. Soit $s = c_0 c_1 \dots c_{n-1}$ un lexème de longueur n, la fonction de hachage est définie par :

$$h(s) = \left(\sum_{i=0}^{n-1} \text{ord}(c_i) \right) \bmod m \tag{33}$$

où $m = 32$ est la taille de la table de hachage et $\text{ord}(c_i)$ est le code ASCII du caractère c_i.

La résolution des collisions se fait par chaînage.

```
[a-zA-Z][a-zA-Z0-9_]*  {
    cdt = cc;
    ltc = yyleng;
    cc += yyleng;
    int num_lex = inserer_lexeme(yytext, yyleng);
    yylval.entier = num_lex;
    return IDF;
}

[0-9]+                 {
    cdt = cc;
    ltc = yyleng;
    cc += yyleng;
    yylval.entier = atoi(yytext);
    return CSTE_ENTIERE;
}

[0-9]+\.[0-9]+         {
    cdt = cc;
    ltc = yyleng;
    cc += yyleng;
    yylval.entier = inserer_lexeme(yytext, yyleng);
    return CSTE_REELLE;
}

'[^']*'                {
    cdt = cc;
    ltc = yyleng;
    cc += yyleng;
    yylval.entier = inserer_lexeme(yytext, yyleng);
    return CSTE_CARACTERE;
}
```

Listing 1 – Exemple de règles lexicales pour identificateurs et constantes

Voir exemple de code complet en Annexe B.1 (page 62).

4.2 Analyse syntaxique

L'analyseur syntaxique est généré par Yacc (parser LALR(1)) à partir d'une grammaire context-free décrivant la syntaxe du langage CPYRR.

4.2.1 Grammaire

La grammaire définit la structure hiérarchique du langage :

— **Programme** : Déclarations de types, variables, fonctions/procédures, puis instructions

— **Déclarations** : Types personnalisés, variables, fonctions, procédures

— **Instructions** : Affectation, conditionnelles, boucles, appels, retour, entrée/sortie

— **Expressions** : Arithmétiques, booléennes, comparaisons, avec gestion de la précédence

4.2.2 Flux de parsing

Le parser LALR(1) fonctionne selon un algorithme de réduction par la gauche avec lookahead :

1. **État initial** : Le parser démarre dans l'état initial avec la pile vide
2. **Shift** : À chaque token reçu, le parser empile le token et passe à un nouvel état selon la table de transitions
3. **Reduce** : Lorsqu'une production complète est reconnue, le parser réduit en remplaçant les symboles de la production par le non-terminal de gauche
4. **Action sémantique** : Lors de la réduction, l'action sémantique associée est exécutée (construction d'AST, ajout dans tables, etc.)
5. **Goto** : Après réduction, le parser effectue une transition selon le non-terminal réduit
6. **Accept** : Lorsque la production racine est réduite, le parsing est accepté

4.2.3 Actions sémantiques

Chaque règle de la grammaire peut avoir des actions sémantiques exécutées lors de la réduction. Voici un exemple pour la construction d'une liste d'instructions :

```
suite_liste_inst
    : instruction POINT_VIRGULE
      {
          /* Creer une liste avec le premier element */
          $$ = ajouter_element_liste(NULL, $1, A_LISTE_INSTRUCTIONS);
      }
    | suite_liste_inst instruction POINT_VIRGULE
      {
          /* Ajouter l'instruction a la liste existante */
          $$ = ajouter_element_liste($1, $2, A_LISTE_INSTRUCTIONS);
      }
    ;

instruction
    : affectation       { $$ = $1; }
    | condition         { $$ = $1; }
    | tant_que          { $$ = $1; }
    | appel             { $$ = $1; }
    | lecture           { $$ = $1; }
    | ecriture          { $$ = $1; }
    | retour            { $$ = $1; }
    ;
```

Listing 2 – Exemple d'actions sémantiques dans le parser (src/analyseur$_{syntaxique/parser.y}$)

Les actions sémantiques permettent de :

— Construire des nœuds AST de manière incrémentale

— Ajouter des déclarations dans les tables pendant le parsing

— Gérer la pile des régions lors de l'entrée/sortie de fonctions

— Propager les types dans les expressions

4.2.4 Gestion des conflits

Les conflits shift/reduce et reduce/reduce sont résolus par :

— Déclaration de précédence et associativité des opérateurs

— Réorganisation de la grammaire pour éliminer les ambiguïtés

— Utilisation de l'option `%define parse.error verbose` pour des messages d'erreur détaillés

Voir exemple de code en Annexe B.2 (page 62).

4.3 Arbre syntaxique abstrait

L'AST constitue la représentation intermédiaire centrale du compilateur. Sa conception permet une séparation claire entre analyse et génération de code, tout en conservant toute l'information nécessaire à l'exécution.

4.3.1 Structure de données

La structure d'un nœud AST est définie comme suit. L'AST peut être modélisé comme un arbre $T = (N, E)$ où N est l'ensemble des nœuds et E l'ensemble des arêtes. Chaque nœud $n \in N$ possède :

— nature$(n) \in \mathcal{N}$: type du nœud (instruction, expression, etc.)

— valeur$(n) \in \mathbb{Z}$: valeur associée

— decl$(n) \in \mathbb{Z} \cup \{-1\}$: numéro de déclaration

— pos$(n) = (\ell, c, L)$: position dans le source (ligne, colonne, longueur)

La structure de l'arbre utilise deux types de chaînage : le **chaînage vertical** via filsGauche(n) qui pointe vers le premier enfant, et le **chaînage horizontal** via frereDroit(n) qui pointe vers le frère suivant.

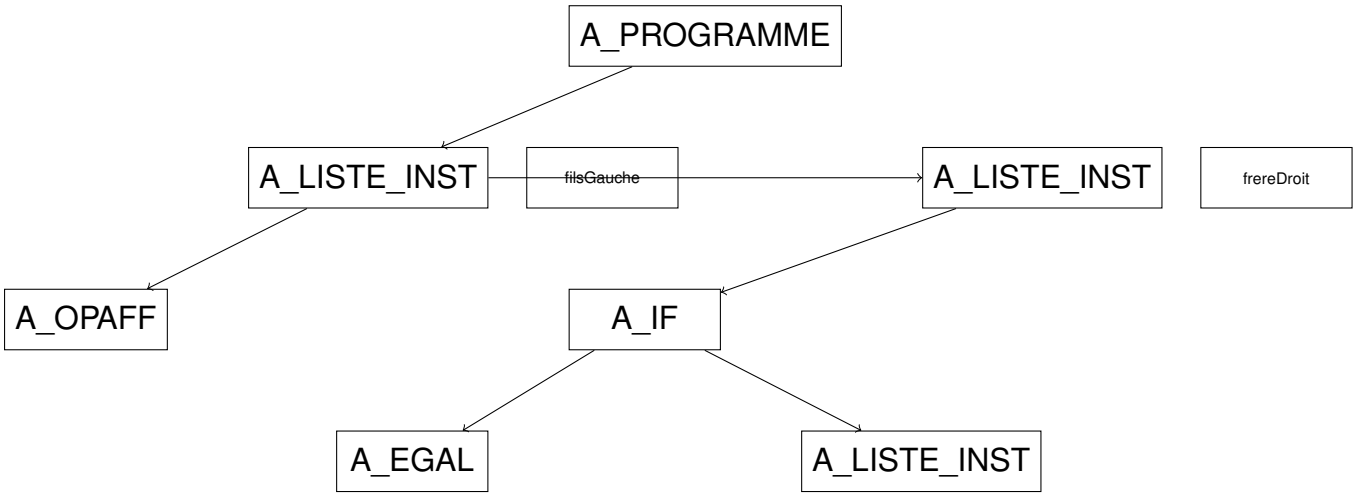

FIGURE 1 – Structure d'un AST avec chaînage vertical (filsGauche) et horizontal (frereDroit)

```c
typedef struct noeud {
    int nature;              // Type du noeud
    int valeur;              // Valeur (num_lex ou constante)
    int num_declaration;     // Index table declarations
    int ligne, colonne, longueur;  // Position source
    struct noeud* filsGauche; // Premier enfant
    struct noeud* frereDroit; // Frere suivant
} noeud;
typedef struct noeud* arbre;
```

Listing 3 – Structure d'un nœud AST (include/ast.h)

4.3.2 Création de nœuds

La création d'un nœud AST se fait via la fonction `creer_noeud` :

```
arbre creer_noeud(int nature, int valeur) {
    arbre n;

    n = (arbre)malloc(sizeof(noeud));
    if (n == NULL) {
        fprintf(stderr, "Erreur : allocation memoire echouee\n");
        exit(EXIT_FAILURE);
    }

    n->nature = nature;
    n->valeur = valeur;
    n->num_declaration = -1;
    n->ligne = ligne_courante;
    n->colonne = cdt;
    n->longueur = ltc;
    n->filsGauche = NULL;
    n->frereDroit = NULL;
    return n;
}
```

Listing 4 – Création d'un nœud AST (src/utils/ast.c)

Pour une description détaillée de la structure et de la manipulation de l'AST, voir la section 5.4 (page 41).

4.4 Tables symboles

Le compilateur utilise quatre tables principales pour gérer les symboles et leurs attributs. Chaque table a un rôle spécifique et des structures de données optimisées pour ses opérations.

4.4.1 Table lexicographique

La table lexicographique stocke tous les identifiants et mots-clés du programme. Elle utilise une table de hachage pour un accès rapide. La structure peut être modélisée comme suit :

$$\mathcal{L} = \{(\ell_i, n_i) \mid i \in [0, N-1]\} \tag{34}$$

où ℓ_i est le lexème et n_i son numéro unique. La fonction de hachage $h : \Sigma^* \to [0, m-1]$ permet d'indexer rapidement les éléments.

Les fonctions de recherche et d'insertion dans la table lexicographique sont détaillées en Annexe B.11 (page 67).

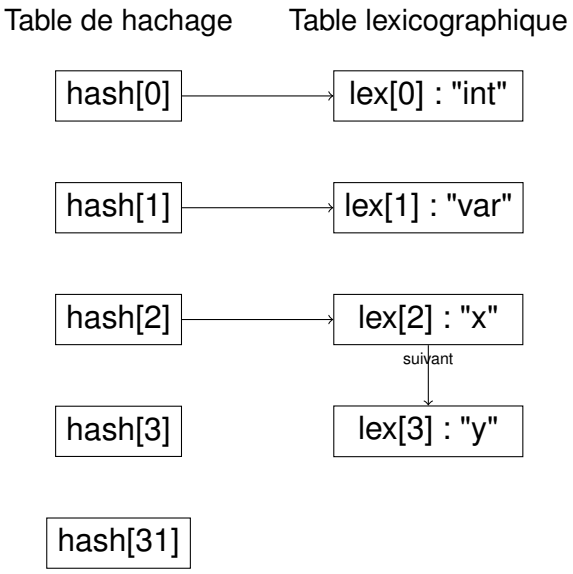

FIGURE 2 – Organisation de la table lexicographique avec hachage et chaînage pour collisions

Pour une description détaillée de l'implémentation de chaque table, voir la section 5.5 (page 45).

4.5 Analyse sémantique

L'analyse sémantique parcourt l'AST construit lors de l'analyse syntaxique et effectue toutes les vérifications de cohérence selon les règles du langage CPYRR.

4.5.1 Vérifications effectuées

L'analyse sémantique effectue un parcours récursif de l'AST pour effectuer toutes les vérifications nécessaires :

1. **Résolution des identificateurs** :
   - Pour chaque identificateur rencontré, recherche dans la pile des régions
   - Parcours de la hiérarchie de portée de la région courante vers la région globale
   - Association du numéro de déclaration au nœud AST si trouvé
   - Génération d'erreur si non déclaré
2. **Vérification des types** :
   - Propagation des types dans les expressions arithmétiques (int, real)
   - Vérification de compatibilité dans les affectations (source et destination)
   - Gestion des conversions implicites (int vers real)
   - Vérification des types dans les comparaisons (types compatibles requis)
3. **Vérification des appels** :
   - Vérification de l'existence de la fonction/procédure
   - Comptage et vérification du nombre de paramètres
   - Vérification du type de chaque paramètre avec la déclaration
   - Pour les fonctions, vérification que le type de retour est utilisé correctement
4. **Vérification des tableaux** :
   - Vérification que le nombre d'indices correspond au nombre de dimensions
   - Vérification que chaque indice est dans les bornes déclarées
   - Vérification que les indices sont de type entier
5. **Vérification des structures** :
   - Vérification que le champ existe dans la structure
   - Vérification de la compatibilité des types lors de l'accès
   - Vérification que l'accès se fait sur une variable de type structure
6. **Détection de warnings** :
   - Parcours de toutes les déclarations pour détecter les variables non utilisées
   - Analyse des conditions pour détecter les conditions toujours vraies/fausses
   - Vérification que les fonctions contiennent au moins un `return`

4.5.2 Parcours de l'AST

L'analyse sémantique parcourt l'AST de manière récursive, en visitant chaque nœud selon sa nature :

— **Instructions** : Parcours séquentiel des listes d'instructions, évaluation des conditions, vérification des affectations

— **Expressions** : Évaluation récursive des sous-expressions, propagation des types de bas en haut

— **Appels** : Vérification des arguments, résolution de la fonction/procédure appelée

— **Accès** : Vérification des variables, tableaux et structures, résolution des identificateurs

4.5.3 Gestion des erreurs

Le système de gestion d'erreurs est un composant fondamental du compilateur, permettant de fournir des messages d'erreur clairs, contextuels et informatifs à l'utilisateur. Il est intégré à toutes les phases de compilation.

Types d'erreurs Le compilateur distingue cinq types d'erreurs selon la phase de détection :

— **Erreurs lexicales** (`ERR_LEXICAL`) : Caractères invalides, lexèmes non reconnus

— **Erreurs syntaxiques** (`ERR_SYNTAXIQUE`) : Violations de la grammaire, tokens inattendus

— **Erreurs sémantiques** (`ERR_SEMANTIQUE`) : Types incompatibles, variables non déclarées, violations de règles sémantiques

— **Warnings** (`ERR_WARNING`) : Avertissements non bloquants (variables non utilisées, etc.)

— **Erreurs d'exécution** (`ERR_RUNTIME`) : Erreurs détectées lors de l'exécution par la machine virtuelle

Structure de données Chaque erreur est stockée dans une structure enrichie contenant :

```
typedef struct {
    TypeErr type;          /* Type d'erreur */
    int ligne;             /* Ligne de l'erreur */
    int colonne;           /* Colonne de d but */
    int longueur;          /* Longueur du token/expression */
    char message[256];     /* Message d'erreur */
    char note[512];        /* Note explicative optionnelle */
} Erreur;
```

Listing 5 – Structure d'une erreur (include/erreurs.h)

Le système maintient un tableau de toutes les erreurs détectées (`tab_erreurs[MAX_ERREURS]`) et un compteur global (`nb_erreurs`).

Collecte des erreurs Les erreurs sont collectées à chaque phase de compilation :

Phase lexicale :

— Détection de caractères invalides

— Utilisation de `ajouter_erreur(ERR_LEXICAL, ligne, ...)` avec position automatique depuis les variables de tracking (`cdt`, `ltc`)

Phase syntaxique :

— Gestion des erreurs de parsing par Yacc avec messages personnalisés

— Utilisation de `yyerror()` pour formater les erreurs syntaxiques

Phase sémantique :

— Vérifications de types, déclarations, portée

— Utilisation de `ajouter_erreur_complete()` pour erreurs avec contexte complet

— Messages détaillés avec notes explicatives

Affichage contextuel Le système d'affichage fournit un contexte riche pour chaque erreur :

1. **En-tête formaté** : `fichier:ligne:colonne: type: message`
2. **Ligne source** : Affichage de la ligne complète avec numérotation
3. **Indicateurs visuels** : Caractères ^ et ~ en couleur verte pointant l'erreur
4. **Note explicative** : Suggestion de correction ou explication supplémentaire

Exemple d'affichage d'erreur :

```
programme.cpyrr:15:8: semantic error: Variable 'x' non d clar e
   15 |     x := 10;
      |        ^
note: V rifiez que la variable est d clar e dans la port e courante
```

Listing 6 – Exemple de sortie d'erreur

Distinction erreurs/warnings Le système distingue clairement les erreurs bloquantes des warnings :

— **Erreurs bloquantes** : Tous les types sauf `ERR_WARNING`
  — Empêchent la génération de code TI
  — Compilation échoue avec code de retour non nul

— **Warnings** : Avertissements non bloquants
  — N'empêchent pas la génération de code
  — Compilation réussit mais affiche les warnings

Les fonctions `compter_erreurs_bloquantes()` et `compter_warnings()` permettent de séparer les deux catégories.

Intégration dans le flux de compilation Le système d'erreurs est intégré à toutes les phases :

```
/* Apr s analyse syntaxique et s mantique */
nb_erreurs_bloquantes = compter_erreurs_bloquantes();
nb_warnings = compter_warnings();

/* Affichage des erreurs */
if (nb_erreurs > 0) {
    afficher_erreurs();
}

/* G n ration TI seulement si pas d'erreurs bloquantes */
if (nb_erreurs_bloquantes == 0) {
    sauvegarder_ti(nom_fichier);
} else {
    printf("    Compilation    choue    : %d erreur(s)\n",
   nb_erreurs_bloquantes);
    exit(EXIT_FAILURE);
}
```

Listing 7 – Intégration dans le point d'entrée (src/main.c)

Gestion du contexte source Pour afficher les lignes de code source, le système charge le fichier source en mémoire :

— `charger_fichier_source()` : Charge toutes les lignes dans un tableau

— `obtenir_ligne_source()` : Récupère une ligne par numéro

— `liberer_fichier_source()` : Libère la mémoire après affichage

Cette approche permet d'afficher le contexte même après la fin de l'analyse.

Indicateurs visuels Les indicateurs visuels utilisent des codes couleur ANSI pour améliorer la lisibilité :

— **Rouge** : Erreurs bloquantes

— **Jaune** : Warnings

— **Vert** : Indicateurs de position (^, ~)

— **Cyan** : Notes explicatives

Le système génère automatiquement les indicateurs selon la colonne et la longueur de l'erreur, permettant de pointer précisément le problème dans le code source.

4.5.4 Orchestration de l'analyse sémantique

L'analyse sémantique est effectuée après la construction complète de l'AST, permettant des vérifications globales :

```
void analyser_semantique() {
    int i;
    extern int compteur_regions;

    /* Verification 1 : Fonctions sans return
     * Pour chaque region de fonction, verifier qu'elle contient au
   moins
     * un A_RETURN. Cette verification necessite un parcours recursif de
     * l'arbre.
     */
    for (i = 1; i < compteur_regions; i++) {
        if (tab_regions[i].numero != -1) {
            verifier_region_fonction(i, tab_regions[i].instructions);
        }
    }

    /* Verification 2 : Variables non utilisees
     * Parcourir toutes les declarations et verifier si chaque variable
     * est utilisee au moins une fois dans l'arbre.
     */
    verifier_variables_non_utilisees();
}
```

Listing 8 – Point d'entrée de l'analyse sémantique (src/semantique/sem$_{a}nalyse.c$)

Voir exemple de code en Annexe B.5 (page 64).

4.6 Machine virtuelle

La machine virtuelle interprète l'AST chargé depuis les fichiers TI. Son architecture repose sur une pile d'exécution organisée en zones, une par région active.

4.6.1 Composants principaux

— **Chargeurs** : Parsing et reconstruction des structures depuis les fichiers TI
— **Pile d'exécution** : Gestion de la mémoire avec chaînage statique
— **Interpréteur** : Évaluation d'expressions et exécution d'instructions
— **Adressage** : Calcul des adresses pour tous les types d'accès

4.6.2 Cycle d'exécution de la machine virtuelle

Le fonctionnement de la machine virtuelle suit un cycle d'exécution bien défini :

1. **Phase de chargement** :
   — Parsing des fichiers TI (lexique.txt, declarations.txt, representations.txt, regions.txt, arbres.txt)
   — Reconstruction des tables en mémoire (table lexicographique, déclarations, représentations, régions)
   — Reconstruction des arbres syntaxiques abstraits par région
   — Validation de la cohérence des données chargées
2. **Initialisation** :
   — Initialisation de la pile d'exécution (toutes les cellules à 0, non initialisées)
   — Positionnement de BC (Base de Code) à 0
   — Positionnement de `region_courante` à 0 (programme principal)
   — Récupération de l'arbre du programme principal (région 0)
3. **Exécution** :
   — Appel récursif de `executer_arbre()` sur l'arbre du programme principal
   — Pour chaque nœud AST, selon sa nature :
      — **Instruction** : Exécution de l'instruction (affectation, condition, boucle, etc.)
      — **Expression** : Évaluation de l'expression et retour de sa valeur
      — **Liste** : Parcours séquentiel des éléments de la liste
   — Gestion des appels de fonctions/procédures avec empilement/dépilement de zones
   — Gestion des retours de fonctions avec dépilement et propagation de la valeur de retour
4. **Terminaison** :
   — Fin de l'exécution lorsque l'arbre principal est complètement parcouru
   — Affichage de l'état final de la pile d'exécution

4.6.3 Initialisation de la pile d'exécution

La pile d'exécution est initialisée au démarrage de la machine virtuelle :

```
void initialiser_pile_execution() {
    int i;

    BC = 0;
    region_courante = 0;

    i = 0;
    while (i < TAILLE_PILE) {
        pile[i].valeur.entier = 0;
        pile[i].est_initialisee = 0;
        i++;
    }
}
```

Listing 9 – Initialisation de la pile d'exécution (src/vm/execution/vm$_{pile.c}$)

4.6.4 Empilement d'une zone

Lors d'un appel de fonction ou de procédure, une nouvelle zone est empilée. Le processus d'empilement est complexe car il doit gérer correctement le chaînage statique selon l'évolution du NIS.

Étapes d'empilement L'empilement d'une zone suit ces étapes :

1. **Calcul de la nouvelle base de code** :

$$\text{BC}_{\text{nouveau}} = \text{BC}_{\text{ancien}} + \text{taille}(\text{region\_courante}) \tag{35}$$

2. **Sauvegarde du BC ancien** : La première case de la nouvelle zone contient le BC de la zone appelante :

$$\text{pile}[\text{BC}_{\text{nouveau}}] = \text{BC}_{\text{ancien}} \tag{36}$$

3. **Gestion du chaînage statique** : Selon l'évolution du NIS, trois cas sont traités (voir algorithme ci-dessous)
4. **Zone de retour** : Si la région appelée est une fonction, une case supplémentaire est réservée pour la valeur de retour :

$$\text{pile}[\text{BC}_{\text{nouveau}} + \text{NIS}_{\text{appele}} + 1] = 0 \quad \text{(non initialisée)} \tag{37}$$

5. **Initialisation des paramètres** : Les valeurs des arguments sont copiées dans les paramètres de la fonction/procédure
6. **Mise à jour de BC et region_courante** : Le pointeur BC est mis à jour et la région courante change

Implémentation détaillée du chaînage statique L'implémentation gère trois cas selon l'évolution du NIS :

```
if (NIS_appele > 0) {
    if (NIS_appele > NIS_appelant) {
        /* Cas 1 : NIS augmente - region appelee plus imbriquee */
        /* Position 1 : BC_ancien */
        pile[BC_nouveau + 1].valeur.entier = BC_ancien;
        pile[BC_nouveau + 1].est_initialisee = 1;

        /* Positions 2    NIS_appele : copier anciens chainages decales */
        for (i = 1; i <= NIS_appelant; i++) {
            pile[BC_nouveau + 1 + i].valeur.entier = pile[BC_ancien + i].valeur.entier;
            pile[BC_nouveau + 1 + i].est_initialisee = 1;
        }
    } else if (NIS_appele == NIS_appelant) {
        /* Cas 2 : NIS stagne - meme niveau d'imbrication */
        copier_chainages_statiques(BC_ancien, BC_nouveau, NIS_appele);
    } else if (NIS_appele < NIS_appelant) {
        /* Cas 3 : NIS decroit - retour vers niveau superieur */
        remontee = NIS_appelant - NIS_appele;
        BC_cible = pile[BC_ancien + remontee].valeur.entier;
        copier_chainages_statiques(BC_cible, BC_nouveau, NIS_appele);
    }
}
```

Listing 10 – Gestion du chaînage statique lors de l'empilement (src/vm/execution/vm$_pile.c$)

Exemple concret d'empilement Considérons un programme avec trois régions :

- Région 0 (globale) : NIS = 0, BC = 0
- Région 1 (fonction f) : NIS = 1, appelée depuis région 0
- Région 2 (fonction g imbriquée dans f) : NIS = 2, appelée depuis région 1

Lors de l'appel de la région 1 depuis la région 0 :

- $\text{BC}_{\text{ancien}} = 0$, $\text{BC}_{\text{nouveau}} = 100$ (par exemple)
- $\text{NIS}_{\text{appelant}} = 0$, $\text{NIS}_{\text{appele}} = 1$ (NIS augmente)
- pile$[100] = 0$ (BC ancien)
- pile$[101] = 0$ (chaînage vers région 0)

Lors de l'appel de la région 2 depuis la région 1 :

- $\text{BC}_{\text{ancien}} = 100$, $\text{BC}_{\text{nouveau}} = 200$
- $\text{NIS}_{\text{appelant}} = 1$, $\text{NIS}_{\text{appele}} = 2$ (NIS augmente)
- pile$[200] = 100$ (BC ancien)
- pile$[201] = 100$ (chaînage vers région 1)
- pile$[202] = 0$ (chaînage vers région 0, copié depuis région 1)

4.6.5 Calcul d'adresse pour tableaux multidimensionnels

Pour un tableau T de dimensions $[b_1..e_1] \times [b_2..e_2] \times \ldots \times [b_n..e_n]$ avec un élément de taille T_{elem}, l'adresse d'un élément $T[i_1][i_2]\ldots[i_n]$ est calculée selon la formule :

$$\text{adr}(T[i_1, \ldots, i_n]) = \text{adr}_{\text{base}} + \sum_{j=1}^{n} e_j \times (i_j - b_j) \tag{38}$$

où les enjambées e_j sont calculées récursivement :

$$e_1 = T_{\text{elem}} \tag{39}$$

$$e_{j+1} = e_j \times (e_j - b_j + 1) \quad \text{pour } j \in [1, n-1] \tag{40}$$

Cette formule garantit un accès efficace en temps constant après calcul des enjambées.

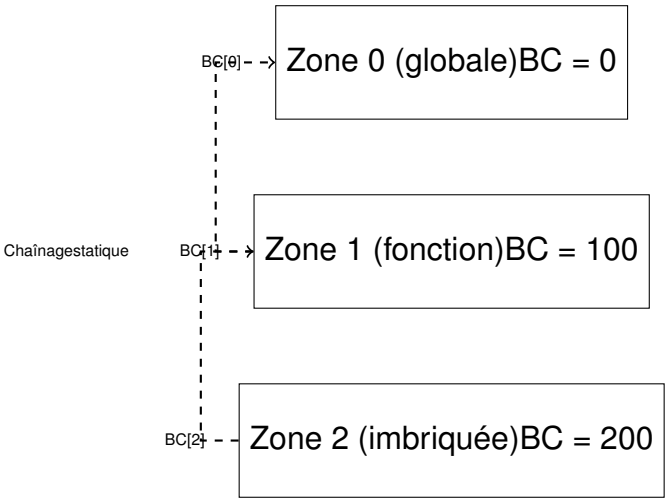

FIGURE 3 – Organisation de la pile d'exécution avec chaînage statique

4.6.6 Adressage avec chaînage statique

Pour une variable déclarée dans une région r_d et utilisée dans une région r_u, l'adresse est calculée via le chaînage statique. Soit NIS(r) le numéro d'index de sauvegarde de la région r, et BC(r) la base de code courante :

$$\text{adr}(v, r_d, r_u) = \text{BC}(\text{remonter}(r_u, \text{NIS}(r_u) - \text{NIS}(r_d))) + \delta(v) + \delta_{\text{retour}}(r_d) \tag{41}$$

où $\delta(v)$ est le déplacement de la variable v dans sa région de déclaration, $\delta_{\text{retour}}(r_d)$ est le décalage dû à la zone de retour si r_d est une fonction, et remonter(r, k) remonte k niveaux dans la chaîne statique.

Fonction de remontée La fonction remonter(r, k) calcule la base de code de la région à k niveaux au-dessus dans la hiérarchie lexicale :

$$\text{remonter}(r, k) = \begin{cases} \text{BC} & \text{si } k = 0 \\ \text{pile}[\text{BC} + k] & \text{si } k > 0 \end{cases} \tag{42}$$

Cette fonction utilise directement le chaînage statique stocké dans la pile d'exécution.

Exemple de calcul d'adresse Considérons une variable x déclarée dans la région 0 (globale) et utilisée dans la région 2 (fonction imbriquée) :

— $r_d = 0$, $r_u = 2$

— NIS$(r_d) = 0$, NIS$(r_u) = 2$

— $k = \text{NIS}(r_u) - \text{NIS}(r_d) = 2$

— $\text{BC}_{\text{cible}} = \text{pile}[\text{BC}_2 + 2] = \text{pile}[202] = 0$ (région globale)

— $\text{adr}(x) = 0 + \delta(x) + 0 = \delta(x)$

Pour une variable y déclarée dans la région 1 et utilisée dans la région 2 :

— $r_d = 1$, $r_u = 2$

— NIS$(r_d) = 1$, NIS$(r_u) = 2$

— $k = 2 - 1 = 1$

— $\text{BC}_{\text{cible}} = \text{pile}[\text{BC}_2 + 1] = \text{pile}[201] = 100$ (région 1)

— $\text{adr}(y) = 100 + \delta(y) + \delta_{\text{retour}}(1)$

4.6.7 Évaluation d'expressions

L'évaluation d'expressions suit un parcours récursif de l'AST des expressions :

— **Constantes** : Retour direct de la valeur (entier, réel, booléen, caractère)

— **Identificateurs** : Calcul de l'adresse, lecture de la valeur depuis la pile

— **Opérations arithmétiques** : Évaluation récursive des opérandes gauche et droit, application de l'opération

— **Opérations booléennes** : Évaluation avec court-circuit pour `&&` et `||`

— **Comparaisons** : Évaluation des opérandes, comparaison, retour d'un booléen

— **Accès tableaux** : Calcul de l'adresse selon les indices, lecture de la valeur

— **Accès structures** : Calcul de l'adresse du champ, lecture de la valeur

— **Appels de fonctions** : Empilement de zone, exécution du corps, récupération de la valeur de retour

4.6.8 Exécution d'instructions

L'exécution d'instructions suit un pattern spécifique selon le type d'instruction :

— **Affectation (`A_OPAFF`)** :

1. Évaluation de l'expression source
2. Calcul de l'adresse de la destination
3. Vérification de compatibilité des types
4. Écriture de la valeur dans la pile à l'adresse calculée
5. Marquage de la cellule comme initialisée

— **Conditionnelle (`A_IF_THEN_ELSE`)** :

1. Évaluation de la condition (retourne un booléen)
2. Si condition vraie : exécution de la branche `then`

3. Sinon : exécution de la branche `else` (si présente)

— **Boucle (`A_WHILE`)** :
1. Évaluation de la condition
2. Tant que condition vraie :
   — Exécution du corps de la boucle
   — Réévaluation de la condition
3. Sortie si condition fausse ou si un `return` est exécuté

— **Appel de procédure (`A_APPEL_PROC`)** :
1. Résolution de la procédure appelée
2. Évaluation des arguments dans l'ordre
3. Empilement d'une nouvelle zone pour la procédure
4. Copie des valeurs des arguments dans les paramètres
5. Exécution du corps de la procédure
6. Dépilement de la zone

— **Appel de fonction (`A_APPEL_FCT`)** :
1. Même processus que pour une procédure
2. Récupération de la valeur de retour depuis la zone de retour
3. Retour de la valeur pour utilisation dans une expression

— **Retour (`A_RETURN`)** :
1. Évaluation de l'expression de retour (si présente)
2. Écriture de la valeur dans la zone de retour
3. Marquage du flag de retour pour sortie anticipée

— **Lecture (`A_LIRE`)** :
1. Parcours de la liste des variables à lire
2. Pour chaque variable : calcul de l'adresse, lecture depuis stdin, écriture dans la pile

— **Écriture (`A_ECRIRE`)** :
1. Parcours de la liste des éléments à écrire
2. Pour chaque élément : évaluation (si expression) ou lecture (si variable), formatage et affichage

Pour une description détaillée de la pile d'exécution et des mécanismes d'adressage, voir la section 5.6 (page 49).

4.7 Représentation intermédiaire

Le compilateur génère une représentation intermédiaire sous forme de fichiers texte structurés. Cette approche permet une séparation complète entre compilation et exécution.

4.7.1 Format des fichiers TI

Le compilateur produit 5 fichiers texte dans un dossier dédié :

1. **lexique.txt** : Table lexicographique (identifiants et mots-clés)
2. **declarations.txt** : Table des déclarations (nature, région, description, exécution)
3. **representations.txt** : Représentations linéaires des types complexes
4. **regions.txt** : Table des régions (NIS, taille, parent)
5. **arbres.txt** : Arbres syntaxiques abstraits par région

4.7.2 Chargement

La machine virtuelle utilise des analyseurs Lex/Yacc dédiés pour parser les fichiers TI et reconstruire les structures en mémoire. Cette approche garantit la validation de la structure et la cohérence des données.

Le processus de chargement suit ces étapes :

1. **Chargement de la table lexicographique** (`lexique.txt`) :
   - Parsing ligne par ligne des lexèmes
   - Reconstruction de la table `tab_lexico` avec tous les identifiants
   - Reconstruction de la table de hachage associée
2. **Chargement de la table des déclarations** (`declarations.txt`) :
   - Parsing des déclarations avec leur nature, région, description, exécution
   - Reconstruction de la table `tab_declarations`
   - Reconstruction des chaînages pour les déclarations multiples
3. **Chargement de la table des représentations** (`representations.txt`) :
   - Parsing des représentations linéaires des types complexes
   - Reconstruction de la table `tab_representation`
   - Calcul des déplacements pour les structures
4. **Chargement de la table des régions** (`regions.txt`) :
   - Parsing des régions avec NIS, taille, parent
   - Reconstruction de la table `tab_regions`
   - Reconstruction de la hiérarchie parent-enfant
5. **Chargement des arbres** (`arbres.txt`) :
   - Parsing récursif des arbres syntaxiques par région
   - Reconstruction des nœuds AST avec leurs propriétés
   - Reconstruction des chaînages vertical (filsGauche) et horizontal (frereDroit)
   - Association des arbres aux régions correspondantes
6. **Validation** :
   - Vérification de la cohérence entre les tables
   - Vérification que toutes les références sont valides
   - Vérification de l'intégrité des arbres

4.7.3 Manipulation des listes dans l'AST

Les listes (instructions, arguments, indices) sont gérées via une fonction spécialisée qui maintient la structure chaînée :

```
arbre ajouter_element_liste(arbre liste_existante, arbre nouvel_element,
    int nature_liste) {
    arbre nouveau_liste, curseur;

    /* Cas 1: Premier element */
    if (liste_existante == NULL) {
        return concat_pere_fils(creer_noeud(nature_liste, -1),
   nouvel_element);
    }

    /* Cas 2: Ajouter a une liste existante */
    nouveau_liste = concat_pere_fils(creer_noeud(nature_liste, -1),
   nouvel_element);

    /* Parcourir la chaine horizontale pour trouver le dernier element
   */
    curseur = liste_existante->filsGauche;

    while (curseur != NULL && curseur->frereDroit != NULL) {
        if (curseur->frereDroit->nature == nature_liste) {
            curseur = curseur->frereDroit->filsGauche;
        } else {
            curseur = curseur->frereDroit;
        }
    }

    /* Accrocher le nouveau A_LISTE comme frere du dernier element */
    if (curseur != NULL) {
        curseur->frereDroit = nouveau_liste;
    }

    return liste_existante;
}
```

Listing 11 – Ajout d'un élément à une liste (src/utils/ast.c)

Voir exemple de format TI en Annexe B.7 (page 64).

4.8 Structures de données et algorithmes avancés

Cette section présente une vue d'ensemble des structures de données et algorithmes complexes utilisés dans l'implémentation du compilateur. Pour une description détaillée de chaque composant, voir les sections 5.4, 5.5 et 5.6.

4.8.1 Structures de données principales

Le compilateur repose sur plusieurs structures de données interconnectées qui permettent de gérer efficacement les symboles, les types et l'exécution :

Arbre syntaxique abstrait (AST) :

- Structure arborescente avec chaînage vertical (`filsGauche`) et horizontal (`frereDroit`)
- 38 types de nœuds différents pour représenter toutes les constructions du langage
- Métadonnées de position (ligne, colonne, longueur) pour le diagnostic d'erreurs
- Lien avec les tables symboles via `num_declaration`
- Gestion efficace des listes (instructions, arguments, indices) via chaînage horizontal

Tables symboles :

- **Table lexicographique** : Stockage de tous les identifiants avec hachage pour accès $O(1)$ amorti
- **Table des déclarations** : Organisation en zones primaire et de débordement avec chaînage pour déclarations multiples
- **Table des représentations** : Représentation linéaire des types complexes (structures, tableaux)
- **Table des régions** : Hiérarchie des portées avec NIS (Numéro d'Index de Sauvegarde) et chaînage parent-enfant

Piles d'exécution :

- **Pile des régions** (compilation) : Gestion de la portée lexicale pendant l'analyse
- **Pile d'exécution** (machine virtuelle) : Zones d'activation avec chaînage statique pour accès aux variables parentes
- Organisation optimisée pour l'empilement/dépilement en $O(NIS)$

Système de gestion d'erreurs :

- Table d'erreurs avec structure enrichie (type, position, message, note)
- Collecte d'erreurs à toutes les phases (lexicale, syntaxique, sémantique, runtime)
- Affichage contextuel avec indicateurs visuels et lignes source
- Distinction erreurs bloquantes/warnings avec comptage séparé
- Gestion du contexte source pour affichage post-analyse

4.8.2 Algorithmes avancés

Fonction de hachage pour table lexicographique :

— Algorithme : somme des codes ASCII des caractères modulo 32
— Résolution des collisions par chaînage séparé
— Complexité : $O(1)$ amorti pour recherche et insertion
— Garantie d'unicité : un même lexème a toujours le même numéro

Chaînage statique pour gestion de portée :

— Algorithme d'empilement adaptatif selon l'évolution du NIS (3 cas : augmentation, stagnation, décroissance)
— Copie intelligente des chaînages selon la profondeur lexicale
— Calcul d'adresse via remontée dans la chaîne statique : $O(\text{NIS}(r_u) - \text{NIS}(r_d))$
— Optimisation : évite la recherche linéaire dans toutes les régions

Calcul d'adresse pour types complexes :

— **Tableaux multidimensionnels** : Calcul d'adresse linéaire avec formule de déplacement
— **Structures** : Accès aux champs via déplacements précalculés dans la table des représentations
— **Chaînage** : Remontée dans la hiérarchie lexicale via chaînage statique
— Complexité : $O(1)$ pour accès direct, $O(d)$ pour remontée de d niveaux

Parcours et manipulation de l'AST :

— Parcours récursif pour analyse sémantique et génération de code
— Construction incrémentale pendant le parsing avec actions sémantiques
— Gestion des listes via insertion en fin de chaîne horizontale
— Sauvegarde/chargement récursif avec format texte structuré

Représentation linéaire des types :

— Structures : séquence de champs avec déplacements calculés
— Tableaux : bornes inférieures/supérieures et tailles de dimensions
— Types composés : représentation récursive avec pointeurs vers sous-types
— Calcul de taille totale en $O(n)$ où n est le nombre de composants

4.8.3 Optimisations et complexité

Les structures de données et algorithmes ont été conçus pour optimiser les opérations fréquentes :

— **Recherche de symboles** : $O(1)$ amorti grâce au hachage
— **Accès aux variables** : $O(d)$ où d est la différence de profondeur lexicale
— **Empilement de zone** : $O(NIS)$ pour copie des chaînages
— **Parcours AST** : $O(n)$ où n est le nombre de nœuds
— **Analyse complète** : $O(n)$ linéaire en taille du programme source

Pour une description détaillée de l'implémentation de chaque structure et algorithme, voir les sections 5.4 (page 41), 5.5 (page 45) et 5.6 (page 49).

5 Résultats et évaluation

5.1 Fonctionnalités implémentées

5.1.1 Couverture du langage CPYRR

Le compilateur supporte l'ensemble des constructions du langage CPYRR :
Types de données :

— Types de base : `int`, `real`, `char`, `bool`, `string`
— Types personnalisés (alias de types)
— Structures avec champs typés
— Tableaux multidimensionnels avec bornes explicites

Instructions :

— Affectation avec vérification de type
— Structures conditionnelles (`if/then/else`)
— Boucles (`while/do`)
— Appels de fonctions et procédures
— Retour de fonction (`return`)
— Entrée/Sortie (`read`/`write` avec formatage)

Expressions :

— Arithmétiques : opérateurs binaires (+, -, *, /) et unaire (-)
— Booléennes : && (ET), || (OU), ! (NON)
— Comparaisons : =, <>, <, >, <=, >=
— Accès complexes : variables, tableaux, champs de structures, chaînage

5.1.2 Qualité de la compilation

Le compilateur fournit :

— Messages d'erreur contextuels avec localisation précise
— Distinction entre erreurs bloquantes et warnings
— Affichage du code source avec indicateurs visuels
— Validation complète de la cohérence sémantique

5.2 Performance et limites

5.2.1 Métriques

— **Taille des tables** : MAX_LEXEMES = 500, MAX_DECLARATIONS = 5000, MAX_REPRESENT
= 10000
— **Taille de la pile** : TAILLE_PILE = 5000 cellules
— **Nombre de régions** : MAX_REGIONS = 100
— **Complexité** : Analyse en $O(n)$ où n est la taille du programme source

5.2.2 Limitations connues

— Pas d'optimisations (propagation de constantes, élimination de code mort)
— Pas de génération de code machine
— Gestion mémoire simplifiée (pas de garbage collector)
— Pas de support de modules ou imports

5.3 Interface utilisateur

5.3.1 Compilateur

```
./bin/compilateur [OPTIONS] fichier.cpyrr

Options:
  -t    Afficher les tables (lexicographique, declarations, etc.)
  -h    Afficher l'aide
```

Listing 12 – Interface en ligne de commande du compilateur

5.3.2 Machine virtuelle

```
./bin/vm_interprete <dossier_ti>

Exemple:
  ./bin/vm_interprete programme_ti
```

Listing 13 – Interface en ligne de commande de la machine virtuelle

5.4 Arbre syntaxique abstrait : structure et manipulation

L'arbre syntaxique abstrait est le cœur de la représentation intermédiaire du compilateur. Cette section détaille sa structure, sa construction et ses mécanismes de manipulation.

5.4.1 Structure d'un nœud AST

Chaque nœud de l'AST est représenté par la structure suivante :

```
typedef struct noeud {
    int nature;              // Type du noeud (instruction, expression, etc.)
    int valeur;              // Valeur (num_lex pour IDF, valeur pour constantes)
    int num_declaration;     // Index dans table des declarations
    int ligne;               // Ligne source pour erreurs
    int colonne;             // Colonne source pour erreurs
    int longueur;            // Longueur du token
    struct noeud* filsGauche; // Premier enfant (ou element de liste)
    struct noeud* frereDroit; // Frere suivant (chainage horizontal)
} noeud;
```

Listing 14 – Structure complète d'un nœud AST

Caractéristiques de la structure :

— **Chaînage vertical** : `filsGauche` pointe vers le premier enfant

— **Chaînage horizontal** : `frereDroit` permet de chaîner les éléments d'une liste

— **Métadonnées de position** : Ligne, colonne et longueur pour localiser précisément les erreurs

— **Lien avec tables** : `num_declaration` relie le nœud à la table des déclarations

5.4.2 Types de nœuds

Le compilateur définit 38 types de nœuds différents, organisés en catégories :
Instructions (1-14) :

— `A_PROGRAMME` : Racine du programme

— `A_LISTE_INSTRUCTIONS` : Liste chaînée d'instructions

— `A_OPAFF` : Affectation (`:=`)

— `A_IF_THEN_ELSE` : Conditionnelle

— `A_WHILE` : Boucle tant que

— `A_APPEL_PROC`, `A_APPEL_FCT` : Appels

— `A_RETURN`, `A_LIRE`, `A_ECRIRE` : Instructions spéciales

Expressions arithmétiques (15-19) :

— `A_PLUS`, `A_MOINS`, `A_MULT`, `A_DIV`

— `A_MOINS_UNAIRE` : Négation arithmétique

Expressions booléennes (20-22) :

— `A_ET`, `A_OU`, `A_NON`

Comparaisons (23-28) :

— `A_EGAL`, `A_DIFF`, `A_INF`, `A_SUP`, etc.

Accès et constantes (29-38) :

— `A_IDF` : Identificateur

— `A_ACCES_TABLEAU` : Accès tableau `tab[i]`

— `A_ACCES_CHAMP` : Accès champ `struct.champ`

— `A_CSTE_ENT`, `A_CSTE_REELLE`, etc. : Constantes

5.4.3 Construction de l'AST

La construction de l'AST se fait de manière incrémentale pendant l'analyse syntaxique :

Création d'un nœud :

```
arbre creer_noeud(int nature, int valeur) {
    arbre n = (arbre)malloc(sizeof(noeud));
    n->nature = nature;
    n->valeur = valeur;
    n->num_declaration = -1;
    n->ligne = ligne_courante;
    n->colonne = cdt;
    n->longueur = ltc;
    n->filsGauche = NULL;
    n->frereDroit = NULL;
    return n;
}
```

Listing 15 – Création d'un nœud AST

Attachement père-fils :

— `concat_pere_fils(pere, fils)` : Attache `fils` comme premier enfant de `pere`

— Utilisé pour construire la hiérarchie verticale

Attachement frère :

— `concat_pere_frere(pere, frere)` : Attache `frere` comme frère droit de `pere`

— Utilisé pour chaîner les éléments d'une liste

5.4.4 Gestion des listes

Les listes (instructions, arguments, indices) sont représentées par une structure récursive :

— Un nœud `A_LISTE_*` contient le premier élément en `filsGauche`

— Les éléments suivants sont chaînés horizontalement via `frereDroit`

— La fonction `ajouter_element_liste()` gère l'ajout à une liste existante

Exemple : Liste d'instructions

```
A_LISTE_INSTRUCTIONS
  +- A_OPAFF (instruction 1)
  |    +- A_IDF (variable)
  |    +- A_PLUS (expression)
  |         +- A_CSTE_ENT (3)
  |         +- A_CSTE_ENT (5)
  +- A_OPAFF (instruction 2)
       +- ...
```

Listing 16 – Structure d'une liste d'instructions

5.4.5 Accès complexes : tableaux et structures

Les accès complexes (tableaux, champs) sont représentés par des chaînes d'accès :

Accès tableau : `tab[i][j]`

— Nœud `A_IDF` pour `tab`

— Enfant : `A_LISTE_INDICES` contenant les indices chaînés

— Chaque indice est une expression évaluable

Accès champ : `point.x`

— Nœud `A_IDF` pour `point`

— Enfant : `A_ACCES_CHAMP` avec `valeur` = num_lex du champ

Accès chaîné : `tab[i].champ`

— Combinaison d'accès tableau et champ

— Géré par `ajouter_liste_acces()`

5.4.6 Parcours et affichage

L'affichage de l'AST utilise un parcours en profondeur récursif :

— Affichage indenté selon la profondeur

— Affichage de la nature, valeur, et métadonnées de chaque nœud

— Supporte l'affichage complet ou partiel (par région)

5.4.7 Sauvegarde de l'AST

L'AST est sauvegardé dans `arbres.txt` avec le format suivant :

```
REGION 0:
NODE A_PROGRAMME 0 -1 0
  NODE A_LISTE_INSTRUCTIONS -1 -1 2
    NODE A_OPAFF -1 -1 2
      NODE A_IDF 5 2 0
      NODE A_PLUS -1 -1 2
        NODE A_CSTE_ENT -1 -1 0
        NODE A_CSTE_ENT -1 -1 0
```

Listing 17 – Format de sauvegarde d'un arbre

Format : `NODE nature num_lex num_decl nb_enfants`

5.4.8 Chargement de l'AST

Le chargement utilise une pile pour reconstruire l'arbre :

— Pile de nœuds en attente d'enfants

— Pour chaque nœud lu, attachement aux parents en attente

— Gestion du nombre d'enfants attendus

5.5 Les tables symboles : organisation et implémentation

Le compilateur utilise quatre tables principales pour gérer les symboles et leurs attributs. Cette section détaille l'implémentation de chacune.

5.5.1 Table lexicographique

Structure :

```
typedef struct {
    char lexeme[MAX_LONGUEUR_LEXEME];
    int longueur;
    int suivant;  // Chainage pour collisions
} Lexeme;

Lexeme tab_lexico[MAX_LEXEMES];  // MAX_LEXEMES = 500
int nb_lexemes;
```

Listing 18 – Structure d'un lexème

Fonctionnement :

— Stocke tous les identifiants et mots-clés rencontrés

— Chaque lexème a un numéro unique (index dans la table)

— Les types prédéfinis (`int`, `real`, `bool`, `char`) sont pré-remplis aux indices 0-3

Table de hashage associée :

— `tab_hashage[TAILLE_HASHAGE]` : Taille 32

— Fonction de hash : somme des caractères modulo 32

— Chaînage pour résoudre les collisions

— Accès en $O(1)$ amorti pour la recherche

Opérations :

— `rechercher_lexeme(lexeme, longueur)` : Recherche avec hash

— `inserer_lexeme(lexeme, longueur)` : Insertion si absent

— Retourne toujours le même numéro pour un même lexème (pas de doublon)

5.5.2 Table des déclarations

Structure :

```
typedef enum {
    TYPE_BASE = 0,     // Types de base (int, real, etc.)
    TYPE_STRUCT = 1,   // Type structure
    TYPE_ARRAY = 2,    // Type tableau
    NATURE_VAR = 3,    // Variable
    NATURE_PARAM = 4,  // Parametre
    NATURE_PROC = 5,   // Procedure
    NATURE_FCT = 6     // Fonction
} Nature;

typedef struct {
    Nature nature;
    int suivant;       // Chainage pour declarations multiples
    int region;        // Numero de region
    int description;   // Index dans table representations
    int execution;     // Numero de region d'execution
} Declaration;

Declaration tab_declarations[MAX_DECLARATIONS];  // MAX_DECLARATIONS =
    5000
```

Listing 19 – Structure d'une déclaration

Organisation en deux zones :

— **Zone primaire** [0..MAX_LEXEMES-1] : Une case par lexème possible

— **Zone de débordement** [MAX_LEXEMES..MAX_DECLARATIONS-1] : Pour déclarations multiples

Chaînage :

— Si un lexème a plusieurs déclarations (différentes régions), elles sont chaînées

— Le champ `suivant` pointe vers la déclaration suivante

— -1 indique la fin de chaîne

Exemple de chaînage :

```
tab_declarations[10].nature = NATURE_VAR;
tab_declarations[10].region = 0;  // Region globale
tab_declarations[10].suivant = 250;  // Prochaine declaration

tab_declarations[250].nature = NATURE_VAR;
tab_declarations[250].region = 3;  // Region locale
tab_declarations[250].suivant = -1;  // Fin de chaine
```

Listing 20 – Exemple : variable x déclarée dans 2 régions

Opérations principales :

— `ajouter_variable(num_lex, region, type)` : Ajoute une variable

— `chercher_dans_region(num_lex, region, nature)` : Recherche dans une région spécifique

— Vérification de doublons dans la même région

5.5.3 Table des représentations

Cette table stocke la représentation linéaire des types complexes (structures, tableaux, fonctions).

Structure :

```
int tab_representation[MAX_REPRESENTATION];  // MAX_REPRESENTATION =
    10000
int ipcv;  // Indice Prochaine Case Vide

// Points d'entr e pour navigation
typedef struct {
    int index;  // Index de d but
    char type;  // 'S' struct, 'A' array, 'F' fonction, 'P' procedure
} PointEntree;

PointEntree points_entree[MAX_POINTS_ENTREE];
```

Listing 21 – Table des représentations

Format pour une structure :

```
[inc] nb_champs | num_lex_1 | type_1 | depl_1 | num_lex_2 | type_2 |
    depl_2 | ...
```

Listing 22 – Format d'une structure

Exemple pour `struct Point { int x; int y; }` :

- `tab_representation[inc] = 2` (nombre de champs)
- `tab_representation[inc+1] = num_lex("x")`
- `tab_representation[inc+2] = 0` (type int)
- `tab_representation[inc+3] = 0` (deplacement, calcule plus tard)
- `tab_representation[inc+4] = num_lex("y")`
- `tab_representation[inc+5] = 0` (type int)
- `tab_representation[inc+6] = 4` (deplacement = taille de int)

Format pour un tableau :

```
[ina] type_elem | nb_dim | borne_inf_1 | borne_sup_1 | borne_inf_2 |
    borne_sup_2 | ...
```

Listing 23 – Format d'un tableau

Format pour une fonction :

```
[inp] type_retour | nb_params | num_lex_1 | type_1 | num_lex_2 | type_2
    | ...
```

Listing 24 – Format d'une fonction

Calcul des deplacements :

- Les deplacements des champs de structures sont calcules lors de la finalisation
- Fonction `remplir_deplacements_structures()`
- Prend en compte l'alignement memoire (simplifie : taille du type)

5.5.4 Table des régions

Structure :

```
typedef struct {
    int numero;         // Numero unique de la region
    int region_parent; // Region parente (hierarchie)
    int nis;            // Numero d'Index de Sauvegarde
    int taille;         // Taille memoire necessaire
    arbre instructions; // Arbre d'instructions de la region
} info_region;

info_region tab_regions[MAX_REGIONS];  // MAX_REGIONS = 100
```

Listing 25 – Structure d'une région

Hiérarchie des régions :

— Région 0 : Programme principal (pas de parent)

— Régions 1+ : Fonctions/procédures (parent = région appelante)

— Chaque région a un NIS (Numéro d'Index de Sauvegarde) pour l'adressage

Calcul du NIS :

— NIS = nombre de régions parentes à remonter pour accéder aux variables

— Région 0 : NIS = 0

— Fonction locale : NIS = 1 (remonte d'un niveau)

— Fonction imbriquée : NIS = 2, 3, etc.

Calcul de la taille :

— Somme des tailles de toutes les variables locales

— Inclut les paramètres

— Inclut la zone de retour pour les fonctions

5.6 Les piles : gestion de la portée et de l'exécution

Le compilateur utilise deux types de piles : une pour la compilation (pile des régions) et une pour l'exécution (pile d'exécution de la VM).

5.6.1 Pile des régions (compilation)

Cette pile maintient la région courante pendant la compilation.

Structure :

```
int pile_regions[MAX_REGIONS];
int sommet_pile;  // Index du sommet (-1 si vide)
int compteur_regions;  // Compteur pour nouvelles regions
```

Listing 26 – Pile des régions

Opérations :

— `initialiser_pile_regions()` : Initialise et empile la région 0

— `empiler_region(num_region)` : Empile une région

— `depiler_region()` : Dépile et retourne la région

— `obtenir_region_courante()` : Retourne le sommet sans dépiler

— `nouvelle_region()` : Crée et retourne un nouveau numéro de région

Utilisation :

— Lors de l'entrée dans une fonction/procédure : empilement

— Lors de la sortie : dépilement

— La région courante détermine où déclarer les variables

— Utilisée pour la résolution de noms (parcours de la pile)

Exemple d'utilisation :

```
// Debut fonction
nouvelle_region();  // Cree region 1
empiler_region(1);  // Pile : [0, 1]

// Declaration variable locale
ajouter_variable(num_lex_x, obtenir_region_courante(), type_int);
// Variable declaree dans region 1

// Fin fonction
depiler_region();  // Pile : [0]
```

Listing 27 – Exemple : compilation d'une fonction

5.6.2 Pile d'exécution (machine virtuelle)

La pile d'exécution est le cœur de la machine virtuelle. Elle gère la mémoire et les appels de fonctions.

Structure de données

```c
typedef union {
    int entier;
    float reel;
    char booleen;
    char caractere;
} Valeur;

typedef struct {
    Valeur valeur;
    int est_initialisee;  // Flag d'initialisation
} Cellule;

Cellule pile[TAILLE_PILE];  // TAILLE_PILE = 5000
int BC;  // Base Courante (pointeur de pile)
int region_courante;  // Region actuellement executee
```

Listing 28 – Structure de la pile d'exécution

Organisation en zones Chaque région active a sa propre zone dans la pile. Formellement, la pile est une séquence :

$$\mathcal{S} = [Z_0, Z_1, \ldots, Z_n] \tag{43}$$

où chaque zone Z_i correspond à une région r_i et commence à l'index $\text{BC}(r_i)$. Les zones sont contiguës et empilées de manière séquentielle.

Structure détaillée d'une zone d'activation Pour une région r avec $\text{NIS}(r) = n$, la zone d'activation Z_r est organisée comme suit :

$$\text{pile}[\text{BC}(r)] = \text{BC}_{\text{ancien}} \quad \text{(lien dynamique)} \tag{44}$$

$$\text{pile}[\text{BC}(r)+1] = \text{BC}(\text{ancêtre}(r,1)) \quad \text{(chaînage statique niveau 1)} \tag{45}$$

$$\text{pile}[\text{BC}(r)+2] = \text{BC}(\text{ancêtre}(r,2)) \quad \text{(chaînage statique niveau 2)} \tag{46}$$

$$\vdots \tag{47}$$

$$\text{pile}[\text{BC}(r)+n] = \text{BC}(\text{ancêtre}(r,n)) \quad \text{(chaînage statique niveau n)} \tag{48}$$

$$\text{pile}[\text{BC}(r)+n+1] = \begin{cases} \text{retour} & \text{si fonction} \\ \text{param}_1 & \text{sinon} \end{cases} \tag{49}$$

$$\text{pile}[\text{BC}(r)+n+2] = \text{param}_2 \text{ ou } \text{var}_1 \tag{50}$$

$$\vdots \tag{51}$$

Chaînage statique détaillé Le chaînage statique permet d'accéder aux variables des régions parentes selon la hiérarchie lexicale. Pour une région r de profondeur d, le chaînage maintient d pointeurs :

$$\forall i \in [1, \text{NIS}(r)] : \text{pile}[\text{BC}(r) + i] = \text{BC}(\text{ancêtre}(r, i)) \tag{52}$$

Cette organisation garantit qu'une variable déclarée dans une région r_d peut être accédée depuis n'importe quelle région r_u en remontant $\text{NIS}(r_u) - \text{NIS}(r_d)$ niveaux dans le chaînage.

Exemple détaillé de chaînage Considérons un programme avec trois régions :

- **Région 0 (globale)** : BC = 0, NIS = 0
  - Variables globales : `pile[0]`, `pile[1]`, ...
  - Pas de chaînage (région racine)
- **Région 1 (fonction f)** : BC = 100, NIS = 1, appelée depuis région 0
  - `pile[100] = 0` (BC ancien = région 0)
  - `pile[101] = 0` (chaînage statique vers région 0)
  - `pile[102] = ?` (zone retour si fonction, sinon premier paramètre)
  - Variables locales : `pile[103]`, `pile[104]`, ...
- **Région 2 (fonction g imbriquée dans f)** : BC = 200, NIS = 2, appelée depuis région 1
  - `pile[200] = 100` (BC ancien = région 1)
  - `pile[201] = 100` (chaînage statique niveau 1 vers région 1)
  - `pile[202] = 0` (chaînage statique niveau 2 vers région 0, copié depuis région 1)
  - `pile[203] = ?` (zone retour)
  - Variables locales : `pile[204]`, `pile[205]`, ...

Algorithme d'empilement avec gestion des trois cas

```
void empiler_zone(int num_region) {
    int BC_ancien, BC_nouveau, taille_courante;
    int NIS_appelant, NIS_appele, est_fonction, remontee, BC_cible, i;

    BC_ancien = BC;
    taille_courante = tab_regions[region_courante].taille;
    BC_nouveau = BC_ancien + taille_courante;

    /* Sauvegarder BC ancien */
    pile[BC_nouveau].valeur.entier = BC_ancien;
    pile[BC_nouveau].est_initialisee = 1;

    NIS_appelant = tab_regions[region_courante].nis;
    NIS_appele = tab_regions[num_region].nis;

    if (NIS_appele > 0) {
        if (NIS_appele > NIS_appelant) {
            /* Cas 1 : NIS augmente */
            pile[BC_nouveau + 1].valeur.entier = BC_ancien;
            pile[BC_nouveau + 1].est_initialisee = 1;
            for (i = 1; i <= NIS_appelant; i++) {
                pile[BC_nouveau + 1 + i].valeur.entier = pile[BC_ancien + i].valeur.
    entier;
                pile[BC_nouveau + 1 + i].est_initialisee = 1;
            }
        } else if (NIS_appele == NIS_appelant) {
            /* Cas 2 : NIS stagne */
            copier_chainages_statiques(BC_ancien, BC_nouveau, NIS_appele);
        } else if (NIS_appele < NIS_appelant) {
            /* Cas 3 : NIS decroit */
            remontee = NIS_appelant - NIS_appele;
            BC_cible = pile[BC_ancien + remontee].valeur.entier;
            copier_chainages_statiques(BC_cible, BC_nouveau, NIS_appele);
        }
    }

    /* Zone de retour pour fonctions */
    est_fonction = determiner_si_fonction(num_region);
    if (est_fonction) {
        pile[BC_nouveau + NIS_appele + 1].valeur.entier = 0;
        pile[BC_nouveau + NIS_appele + 1].est_initialisee = 0;
    }

    BC = BC_nouveau;
    region_courante = num_region;
}
```

Listing 29 – Algorithme complet d'empilement (src/vm/execution/vm$_{p}ile.c$)

Fonction de copie des chaînages La fonction `copier_chainages_statiques` copie les chaînages d'une zone source vers une zone destination :

```
static void copier_chainages_statiques(int BC_source, int BC_dest,
                                       int nb_chainages) {
    int i = 1;
    while (i <= nb_chainages) {
        pile[BC_dest + i].valeur.entier = pile[BC_source + i].valeur.
   entier;
        pile[BC_dest + i].est_initialisee = 1;
        i++;
    }
}
```

Listing 30 – Fonction de copie des chaînages statiques

Dépilement d'une zone Le dépilement suit un processus inverse :

1. **Récupération de la valeur de retour** : Si la région est une fonction, lire depuis pile$[\text{BC} + \text{NIS} + 1]$
2. **Restauration du BC** : $\text{BC}_{\text{nouveau}} = \text{pile}[\text{BC}_{\text{ancien}}]$
3. **Restauration de la région** : Depuis la pile auxiliaire $\mathcal{P}_{\text{regions}}$
4. **Réinitialisation** : Toutes les variables de la zone sont réinitialisées

```
void depiler_zone() {
    int BC_ancien, BC_nouveau;
    int taille_region, debut_variables, i;

    BC_ancien = BC;
    taille_region = tab_regions[region_courante].taille;
    debut_variables = BC + tab_regions[region_courante].nis + 1;

    /* Reinitialiser les variables de la region depilee */
    i = debut_variables;
    while (i < BC + taille_region) {
        pile[i].valeur.entier = 0;
        pile[i].est_initialisee = 0;
        i++;
    }

    /* Restaurer BC */
    BC_nouveau = pile[BC].valeur.entier;
    BC = BC_nouveau;

    /* Restaurer region depuis la pile auxiliaire */
    if (sommet_pile_regions_vm > 0) {
        sommet_pile_regions_vm--;
        region_courante = pile_regions_vm[sommet_pile_regions_vm];
    } else {
        region_courante = 0;
    }
}
```

Listing 31 – Dépilement d'une zone (src/vm/execution/vm$_{pile}$.c)

Calcul d'adresse avec chaînage statique Pour une variable v déclarée dans r_d et utilisée dans r_u, l'algorithme de calcul d'adresse est :

Input: Numéro de déclaration d, région courante r_u
Output: Adresse mémoire de la variable
$r_d \leftarrow$ region(d);
$\delta \leftarrow$ deplacement(d);
NIS$_d \leftarrow$ NIS(r_d);
NIS$_u \leftarrow$ NIS(r_u);
$k \leftarrow$ NIS$_u -$ NIS$_d$;
if $k = 0$ **then**
  `// Variable locale`
  BC$_d \leftarrow$ BC;
end
else if $k > 0$ **then**
  `// Variable dans une région parente`
  BC$_d \leftarrow$ pile[BC $+ k$];
end
else
  `// Erreur : variable dans une région enfant`
  return *erreur*;
end
if $r_d = 0$ **then**
  `// Région globale`
  adr $\leftarrow \delta$;
end
else
  $\delta_{\text{retour}} \leftarrow$ est_fonction$(r_d)?1 : 0$;
  adr $\leftarrow$ BC$_d + \delta + \delta_{\text{retour}}$;
end
return *adr*;

Algorithm 4: Calcul d'adresse d'une variable

Vérification d'initialisation Chaque cellule de la pile possède un flag `est_initialisee` qui permet de détecter l'utilisation de variables non initialisées. Lors d'une lecture :

$$\text{lecture}(adr) = \begin{cases} \text{pile}[adr].\text{valeur} & \text{si pile}[adr].\text{est\_initialisee} = 1 \\ \text{erreur} & \text{sinon} \end{cases} \tag{53}$$

Cette vérification permet de détecter les bugs d'utilisation avant initialisation à l'exécution.

5.6.3 Complexité et optimisations

Complexité des opérations :

— Empilement/dépilement : $O(NIS)$ (copie des chaînages)

— Accès variable locale : $O(1)$

— Accès variable parente : $O(1)$ (via chaînage)

— Recherche dans pile régions : $O(d)$ où d = profondeur

Optimisations possibles :

— Cache des adresses calculées

— Réduction du nombre de chaînages (optimisation NIS)

— Allocation contiguë des variables locales

6 Conclusion et perspectives

6.1 Outils et bibliothèques

— **GCC** : Compilateur C (gcc)
— **Lex** : Générateur d'analyseurs lexicaux
— **Yacc/Bison** : Générateur d'analyseurs syntaxiques
— **Make** : Système de construction
— **Standard C Library** : stdio.h, stdlib.h, string.h, etc.

6.2 Documentation et références

— Documentation officielle Lex : https://www.gnu.org/software/lex/
— Documentation officielle Yacc/Bison : https://www.gnu.org/software/bison/
— "Compilers : Principles, Techniques, and Tools" (Dragon Book) - Aho, Lam, Sethi, Ullman
— Cours "Compilation" de Licence 3 Informatique, Université Jean Monnet, 2025
— Stack Overflow pour résolution de problèmes spécifiques
— GeeksforGeeks - Articles sur la compilation

6.3 Standards et conventions

— Style de code C (K&R avec adaptations)
— Convention de nommage : snake_case pour fonctions et variables
— Documentation des fonctions avec commentaires
— Gestion d'erreurs cohérente dans tout le projet

6.4 Bilan

Ce projet a permis d'implémenter un compilateur complet pour le langage CPYRR, couvrant toutes les phases classiques de compilation. Les résultats démontrent la faisabilité de l'approche par représentation intermédiaire textuelle et l'efficacité des outils de génération (Lex/Yacc) pour simplifier l'implémentation.

6.4.1 Contributions principales

— Architecture modulaire séparant compilation et exécution
— Système de gestion de portée avec régions et chaînage statique
— Représentation intermédiaire textuelle pour débogage et inspection
— Machine virtuelle complète avec gestion de la mémoire et adressage
— Système de gestion d'erreurs contextuel et informatif

6.4.2 Apports pédagogiques

Ce projet a permis de mettre en pratique :

— Les concepts théoriques de compilation (automates, grammaires, parsing)
— Les techniques de gestion de structures arborescentes (AST)
— Les algorithmes de gestion de la portée et de l'adressage mémoire
— L'utilisation d'outils de génération (Lex/Yacc)
— La conception de systèmes complexes modulaires

6.5 Perspectives d'évolution

Plusieurs axes d'amélioration peuvent être envisagés :

6.5.1 Optimisations

— Propagation de constantes
— Élimination de code mort
— Optimisation des boucles
— Réduction de la taille du code généré

6.5.2 Extensions du langage

— Support de modules et imports
— Système de types avancé (génériques, inférence de types)
— Gestion d'exceptions
— Programmation orientée objet

6.5.3 Amélioration de la machine virtuelle

— Garbage collector pour gestion automatique de la mémoire
— Compilation JIT (Just-In-Time)
— Debugger intégré avec points d'arrêt
— Profiling et analyse de performance

6.5.4 Génération de code

— Génération de code machine (assembleur)
— Génération de bytecode (JVM, .NET)
— Support de plusieurs architectures cibles

7 Références et ressources

7.1 Outils et bibliothèques

— **GCC** : Compilateur C (gcc)

— **Lex** : Générateur d'analyseurs lexicaux

— **Yacc**/**Bison** : Générateur d'analyseurs syntaxiques

— **Make** : Système de construction

— **Standard C Library** : stdio.h, stdlib.h, string.h, etc.

7.2 Documentation et références bibliographiques

— Documentation officielle Lex : https://www.gnu.org/software/flex/

— Documentation officielle Yacc/Bison : https://www.gnu.org/software/bison/

— Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers : Principles, Techniques, and Tools* (2nd ed.). Pearson Education. (Dragon Book)

— Cours "Compilation" de Licence 3 Informatique, Université Jean Monnet, 2025

— Appel, A. W. (1998). *Modern Compiler Implementation in C*. Cambridge University Press

7.3 Standards et conventions

— Style de code C (K&R avec adaptations)

— Convention de nommage : snake_case pour fonctions et variables

— Documentation des fonctions avec commentaires

— Gestion d'erreurs cohérente dans tout le projet

Annexe A : Architecture et diagrammes

Cette annexe présente les diagrammes d'architecture du projet pour faciliter la compréhension de l'organisation globale.

A.1 - Architecture complète du compilateur

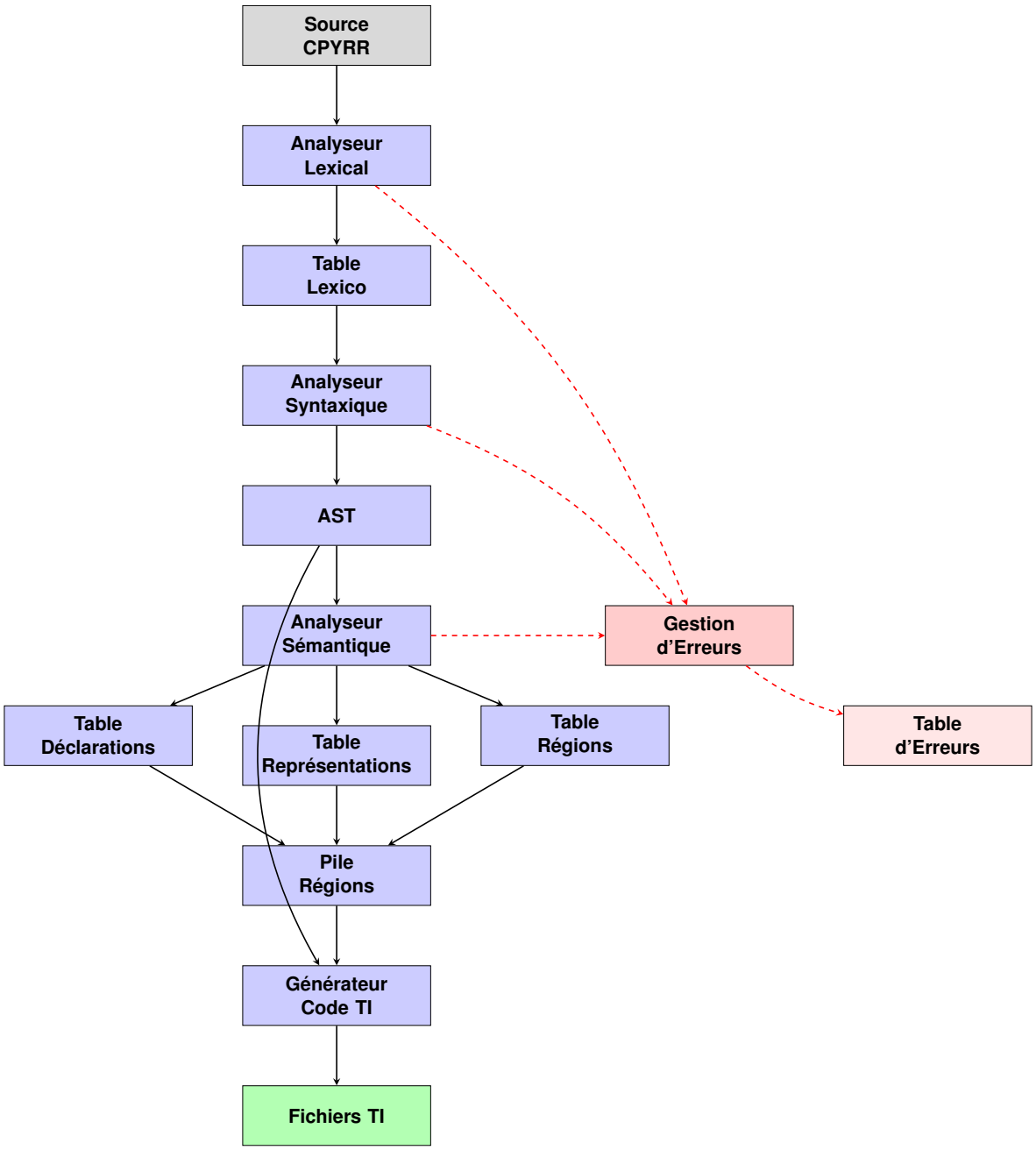

FIGURE 4 – Architecture complète montrant les interactions entre tous les modules

Note : Ce diagramme montre l'organisation complète du compilateur avec les flux de données entre les différentes phases. Le système de gestion d'erreurs (en rouge) collecte les erreurs de toutes les phases et les stocke dans la table d'erreurs.

A.2 - Phases de compilation

Ce diagramme présente les différentes phases de compilation et leurs interactions :

— **Analyse lexicale** : Source → Tokens
— **Analyse syntaxique** : Tokens → AST
— **Analyse sémantique** : AST + Tables → AST annoté
— **Génération TI** : AST + Tables → Fichiers texte

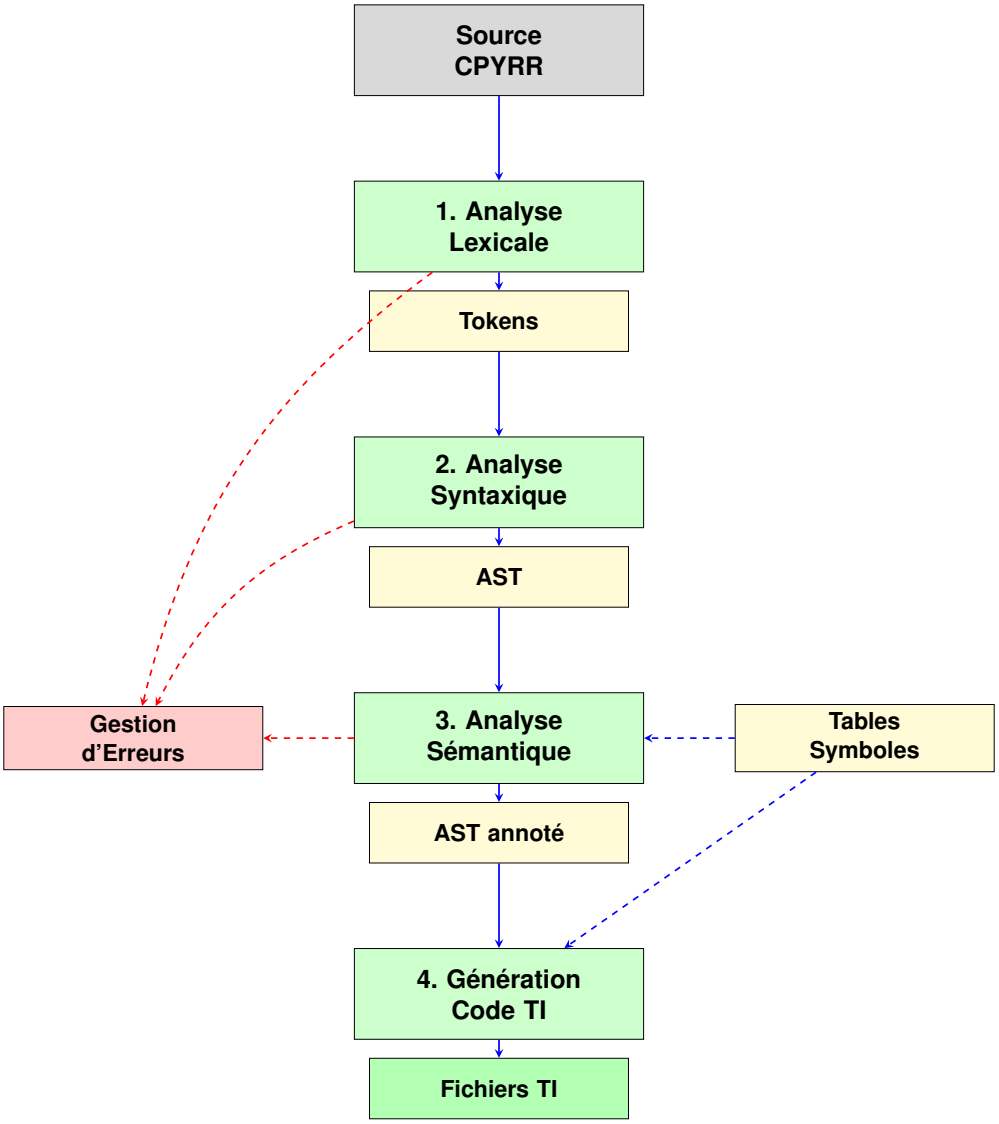

FIGURE 5 – Flux de données à travers les phases de compilation

A.3 - Structure de la machine virtuelle

Ce diagramme présente l'architecture de la machine virtuelle :

— **Chargeurs** : Parsing des fichiers TI
— **Pile d'exécution** : Gestion de la mémoire
— **Interpréteur** : Exécution de l'AST
— **Utilitaires** : Expressions, instructions, adressage

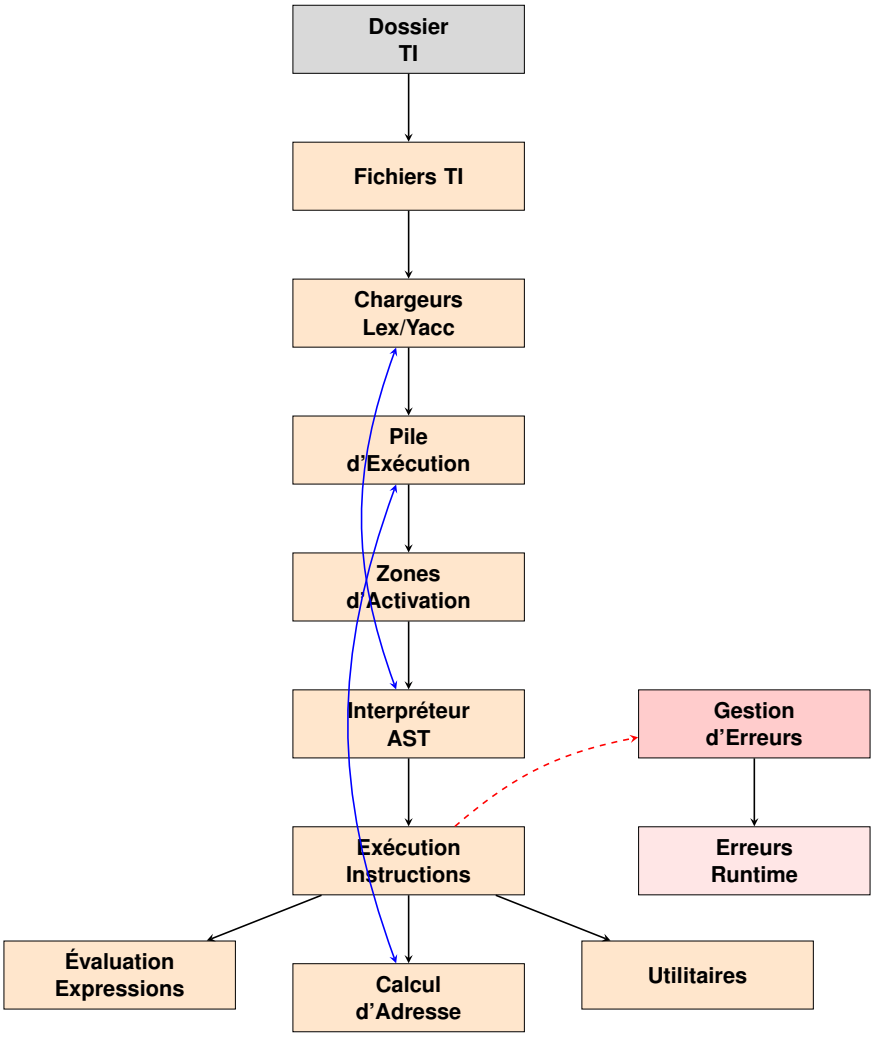

FIGURE 6 – Architecture de la machine virtuelle

Annexe B : Exemples de code

Cette annexe présente des exemples de code illustrant les techniques de compilation décrites dans la section 3.

B.1 - Analyseur lexical (Lex)

```
"prog"                              { cdt = cc; ltc = yyleng; cc += yyleng;
    return PROG; }
"var"                               { cdt = cc; ltc = yyleng; cc += yyleng;
    return VARIABLE; }
"int"                               { cdt = cc; ltc = yyleng; cc += yyleng;
    return ENTIER; }
[0-9]+                              { cdt = cc; ltc = yyleng; cc += yyleng;
                                      yylval.entier = atoi(yytext);
                                      return CSTE_ENTIERE; }
[a-zA-Z][a-zA-Z0-9_]*             { cdt = cc; ltc = yyleng; cc += yyleng;
                                      int num = rechercher_lexeme(yytext,
   yyleng);
                                      if (num == -1) {
                                          num = inserer_lexeme(yytext,
   yyleng);
                                      }
                                      yylval.entier = num;
                                      return IDF; }
```

Listing 32 – Extrait de lexer.l

B.2 - Analyseur syntaxique (Yacc)

```
programme
    : PROG
      {
          initialiser_pile_regions();
          initialiser_tab_declarations();
          initialiser_tab_representation();
      }
      corps
      {
          associer_arbre_region(0, $3);
          finaliser_region_principale();
      }
    ;

declaration_variable
    : VARIABLE IDF DEUX_POINTS nom_type
      {
          ajouter_variable($2,
                           obtenir_region_courante(),
                           $4);
      }
    ;
```

Listing 33 – Extrait de parser.y

B.3 - Structure de l'AST

```c
typedef struct noeud {
    int nature;
    int valeur;
    int num_declaration;
    int ligne;
    int colonne;
    int longueur;
    struct noeud* filsGauche;
    struct noeud* frereDroit;
} noeud;

typedef struct noeud* arbre;

arbre creer_noeud(int nature, int valeur) {
    arbre n = (arbre)malloc(sizeof(noeud));
    n->nature = nature;
    n->valeur = valeur;
    n->filsGauche = NULL;
    n->frereDroit = NULL;
    return n;
}
```

Listing 34 – Structure d'un nœud AST

B.4 - Gestion des tables

```c
typedef struct {
    Nature nature;
    int suivant;
    int region;
    int description;
    int execution;
} Declaration;

extern Declaration tab_declarations[MAX_DECLARATIONS];

int ajouter_variable(int num_lex, int region, int type_index) {
    int index = prochaine_case_libre++;
    tab_declarations[index].nature = NATURE_VAR;
    tab_declarations[index].region = region;
    tab_declarations[index].description = type_index;
    // Chainage...
    return index;
}
```

Listing 35 – Exemple de table des déclarations

B.5 - Analyse sémantique

```
arbre verifier_affectation(NoeudType destination, NoeudType source) {
    int type_dest = destination.type;
    int type_src = source.type;

    if (!types_compatibles(type_dest, type_src)) {
        ajouter_erreur_complete(ERR_SEMANTIQUE,
                                destination.ligne,
                                destination.colonne,
                                destination.longueur,
                                "Types incompatibles dans l'affectation",
                                "Impossible d'affecter un type
   incompatible");
        return NULL;
    }

    return construire_affectation(destination.tarbre, source.tarbre);
}
```

Listing 36 – Vérification de type dans une affectation

B.6 - Machine virtuelle

```
Valeur evaluer_arbre(arbre a) {
    switch (a->nature) {
        case A_PLUS:
            return appliquer_operation_arithmetique(PLUS,
                                                    evaluer_arbre(a->
   filsGauche),
                                                    evaluer_arbre(a->
   filsGauche->frereDroit),
                                                    obtenir_type_noeud(a
   ));
        case A_IDF:
            return evaluer_variable(a);
        case A_CSTE_ENT:
            return (Valeur){.entier = a->valeur};
        // ...
    }
}
```

Listing 37 – Évaluation d'une expression arithmétique

B.7 - Format des fichiers TI

```
nb_declarations: 10
DECL 0: nature=TYPE_BASE region=0 description=0 execution=0
DECL 1: nature=TYPE_BASE region=0 description=1 execution=0
DECL 2: nature=NATURE_VAR region=0 description=0 execution=0
DECL 3: nature=NATURE_FCT region=0 description=5 execution=0
...
```

Listing 38 – Exemple de fichier declarations.txt

B.8 - Gestion des régions

```
int association_nom(int num_lex, Nature nature_recherchee) {
    int region_courante = obtenir_region_courante();

    // Parcours de la pile des regions
    while (region_courante >= 0) {
        int index = chercher_dans_region(num_lex, region_courante,
   nature_recherchee);
        if (index != -1) {
            return index;
        }
        region_courante = obtenir_parent_region(region_courante);
    }

    // Non trouve
    return -1;
}
```

Listing 39 – Résolution d'un identificateur

B.9 - Calcul d'adresse

```
int calculer_adresse_tableau(int adresse_base, int num_representation,
                             int* indices, int nb_indices) {
    int adresse = adresse_base;
    int* bornes_inf = obtenir_bornes_inf(num_representation);
    int* bornes_sup = obtenir_bornes_sup(num_representation);

    for (int i = 0; i < nb_indices; i++) {
        int taille_dim = bornes_sup[i] - bornes_inf[i] + 1;
        adresse += (indices[i] - bornes_inf[i]) * taille_dim;
    }

    return adresse;
}
```

Listing 40 – Calcul d'adresse pour un tableau

B.10 - Point d'entrée principal

```c
int main(int argc, char *argv[]) {
    int opt, nb_erreurs_bloquantes, nb_warnings;
    char *fichier;

    /* Traitement des options avec getopt */
    while ((opt = getopt(argc, argv, "th")) != -1) {
        if (opt == 't') afficher_tables = 1;
        else if (opt == 'h') { afficher_aide(argv[0]); exit(0); }
    }

    /* Recuperer le nom du fichier */
    if (optind < argc) fichier = argv[optind];
    else { fprintf(stderr, "Erreur: aucun fichier specifie\n"); exit(1); }

    /* Ouvrir le fichier */
    nom_fichier = fichier;
    yyin = fopen(nom_fichier, "r");
    if (!yyin) { fprintf(stderr, "Erreur: impossible d'ouvrir %s\n", nom_fichier); exit
    (1); }

    charger_fichier_source(nom_fichier);

    /* Initialisation */
    init_erreurs();
    initialiser_tab_lexico();
    initialiser_pile_regions();
    initialiser_tab_regions();
    initialiser_tab_declarations();
    initialiser_tab_representation();

    /* Analyse syntaxique */
    if (yyparse() == 0) analyser_semantique();

    fclose(yyin);

    /* Compter erreurs vs warnings */
    nb_erreurs_bloquantes = compter_erreurs_bloquantes();
    nb_warnings = compter_warnings();

    /* Generation TI si pas d'erreurs bloquantes */
    if (nb_erreurs_bloquantes == 0) sauvegarder_ti(nom_fichier);

    return (nb_erreurs_bloquantes > 0) ? 1 : 0;
}
```

Listing 41 – Point d'entrée principal du compilateur (src/main.c)

B.11 - Fonctions de recherche et insertion lexicographique

```c
int rechercher_lexeme(const char* lexeme, int longueur) {
    int hash, indice;
    if (longueur < 0) longueur = strlen(lexeme);
    hash = calculer_hashage(lexeme);
    indice = tab_hashage[hash];
    while (indice != -1) {
        if (tab_lexico[indice].longueur == longueur) {
            if (strcmp(tab_lexico[indice].lexeme, lexeme) == 0) return indice;
        }
        indice = tab_lexico[indice].suivant;
    }
    return -1;
}

int inserer_lexeme(const char* lexeme, int longueur) {
    int numero, hash;
    if (longueur < 0) longueur = strlen(lexeme);
    numero = rechercher_lexeme(lexeme, longueur);
    if (numero != -1) return numero;  // Deja present
    if (nb_lexemes >= MAX_LEXEMES) {
        fprintf(stderr, "Erreur : table lexicographique pleine\n");
        return -1;
    }
    hash = calculer_hashage(lexeme);
    numero = nb_lexemes;
    strncpy(tab_lexico[numero].lexeme, lexeme, MAX_LONGUEUR_LEXEME - 1);
    tab_lexico[numero].lexeme[MAX_LONGUEUR_LEXEME - 1] = '\0';
    tab_lexico[numero].longueur = longueur;
    tab_lexico[numero].suivant = tab_hashage[hash];
    tab_hashage[hash] = numero;
    nb_lexemes++;
    return numero;
}
```

Listing 42 – Recherche et insertion dans la table lexicographique (src/tables/tab$_{l}exico.c$)