# Projet de compilation en licence informatique: mini compilateur C vers un langage de graphes

Pr Sid Touati, Université Côte d'Azur

2025

## Table des matières

# 1 Consignes sur la soumission et notation du projet de compilation

Le fichier `"Grille d'évaluation du projet de compilation"` mis en ligne dans l'espace internet de l'UE de compilation indique avec précision quoi faire pour soumettre votre projet et comment celui-ci sera noté. Suivre les consignes demandées fait partie de la notation du projet.

# 2 Présentation générale du projet

Le but de ce projet est de réaliser un compilateur d'un sous langage du C. Votre compilateur prendra en entrée un code C simplifié et génèrera une représentation intermédiaire sous forme de graphes (arbres abstraits). Le langage C simplifié en entrée s'appelle miniC et le langage en sortie s'appelle DOT.

Un code écrit en miniC peut être compilé et testé avec n'importe quel compilateur C comme gcc. Un graphe en format DOT peut être visualisé et traité avec plusieurs outils libres comme `GraphViz`. Il existe aussi des outils libres comme la commande `dot` qui permettent de convertir un fichier DOT vers un fichier pdf : une petite recherche sur internet vous permettra d'en savoir plus.

# 3 Langage source : miniC

Il s'agit d'un sous-ensemble de C qui doit pouvoir être compilé avec un compilateur C classique comme gcc. Il a les caractéristiques suivantes.

## 3.1 Description du langage miniC

- Structure du programme :
  - la structure d'un programme est la suivante :

    *<declarations>*
    *<fonctions>*
  - toute variable doit être déclarée avant utilisation.
- Les identificateurs :
  - ils ne peuvent pas porter le nom d'un mot clé réserve : **extern, int, void, for, while, if, then, else, switch, case, default, break**.
- Les variables :
  - elles ne peuvent être déclarées qu'en début de programme (variables globales) ou qu'au début d'un bloc (variables locales).
  - elles doivent commencer par une lettre,
  - elles sont de type entier ou tableau d'entiers à un nombre quelconque de dimensions.
- Les constantes :
  - elles sont entières et exprimées en base 10.
- Les fonctions :
  - structure d'une fonction :

    *<type> identificateur (<liste_parametres>)*
    *{*
    *<bloc>*
    *}*
  - **identificateur** est le nom de la fonction
  - le **type** d'une fonction est soit **int** soit **void**
  - un seul niveau de déclaration de fonctions est possible (pas d'imbrications de déclarations)
  - les paramètres sont des entiers
  - une fonction peut être déclarée **extern**. Dans ce cas, elle est déclarée mais pas définie/implémentée dans le programme (son code est dans un autre fichier). Cela permet de faire référencer des

fonctions de librairie ou écrites dans un autre fichier tout en permettant de vérifier que toutes les fonctions utilisées soient déclarées.

- Les instructions considérées sont :
  - L'affectation *var* = *expression*
  - les structures de contrôle
  - **return**. Elle peut retourner une valeur ou non.
- Les instructions de contrôle ont la même sémantique que celle du C :
  - l'instruction **if..then** et **if..then..else**
  - l'instruction **switch**, avec **case** et **default**
  - l'instruction **for**
  - l'instruction **while**
  - l'instruction **break**
- Les conditions :
  - une condition est toujours entre parenthèses
  - les opérateurs de comparaison dont on dispose sont : >, <, <=, >=, ==, **!=**
  - les opérateurs booléens sont : **!**, **&&**, ||
- Les expressions :
  - les opérateurs binaires dont on dispose sont : +, **-**, *****, /, «, », &, |
  - une expression est composée de valeurs numériques et/ou de variables et d'opérateurs binaires. Les appels de fonctions peuvent faire partie des expression arithmétiques.
- Les commentaires :
  - les commentaires débutent par /* et se terminent par */
  - ils peuvent être sur plusieurs lignes

### 3.2 Exemple d'un programme écrit en miniC

Le code `exempleminiC.c` est fourni en exemple. Vous pouvez tester que gcc le compile correctement avec la commande `gcc -c exempleminiC.c`, cela produira un code objet.

### 3.3 Grammaire du langage miniC

La grammaire initiale du langage miniC est décrite dans le fichier `Grammaire-miniC.html`. Nous fournissons une description yacc de départ dans le fichier appelé `miniC.y`. Vous pouvez modifier ou compléter cette description yacc pour le besoin de votre projet, en l'indiquant ensuite dans votre rapport. Si vous sentez le besoin de modifier ou d'améliorer la grammaire du langage miniC, vous pouvez le faire à condition que cette modification produise des codes corrects vis à vis de la syntaxe du C. Mais attention, assurez-vous que votre fichier yacc final ne comporte pas de conflits *shift-reduce*.

Également, nous fournissons une description lex du langage C dans le fichier nommé `ANSI-C.l`. Vous pouvez modifier ou compléter cette description lex pour la rendre correcte vis à vis du langage miniC ; par exemple, les mots clés du langage C qui n'appartiennent pas au langage miniC doivent être enlevés, etc. Il est à rappeler que l'analyseur lexical ne doit renvoyer que les tokens valides vers l'analyseur syntaxique, tout autre token qui n'appartient pas au langage considéré doit être négligé.

## 4 Langage destination : DOT

### 4.1 Description du langage DOT

DOT est un langage de description de graphe dans un format texte. Il fait partie de l'ensemble d'outils libres `Graphviz`. La description DOT d'un graphe est assez riche pour contenir toutes les informations sur un graphe quelconque : attributs des noeuds, couleurs, formes géométriques, étiquettes, disposition graphique

(*layout*) , etc. Vous devrez utiliser ce langage pour générer dans un fichier la représentation intermédiaire du programme sous forme d'arbres abstraits. Une recherche sur internet vous permettra d'avoir plus de détails sur le langage DOT, le document courant donnera quelques éléments de base.

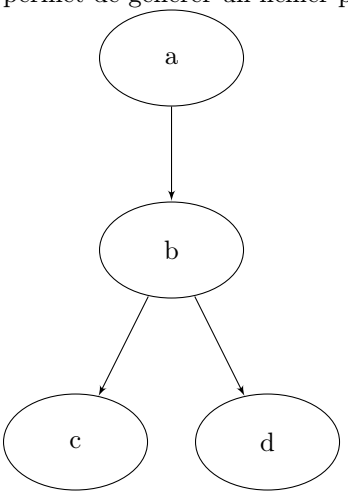
Le langage DOT traite à la fois les graphes orientés et les graphes non orientés. Dans ce projet, nous utiliserons uniquement les graphes orientés.

## 4.2 Un exemple simple de fichier DOT

La syntaxe DOT peut décrire des graphes orientés. Le mot clé de création du graphe est **digraph** (*directed graph*). Les arcs sont décrits par une flèche (`->`). Ci-dessous un exemple simple d'un fichier DOT. Les commentaires sont précédés par `//`.

```
digraph mon_graphe {
   // Ceci est un commentaire
    a -> b -> c;
    b -> d;
}
```

Un fichier dot peut être visualisé avec un logiciel graphique interactif (il y en a plusieurs disponibles), ou traité avec des commandes de la collection `GraphViz`. Par exemple, la commande `dot -Tpdf ex.dot -o ex.pdf` permet de générer un fichier pdf à partir d'un fichier DOT. La visualisation du graphe donnera ce qui suit :




Plusieurs attributs peuvent être associés aux graphes, nœuds et arcs dans un fichier DOT. Ces attributs contrôlent des aspects comme la couleur, la forme et le style de trait. Pour les nœuds et arcs, les attributs sont placés entre crochets (`[ ]` ). Un attribut est spécifié sous la forme `[`nomattribut=valeur`]`; il est possible de spécifier plusieurs attributs en les séparant par des blancs comme ceci :

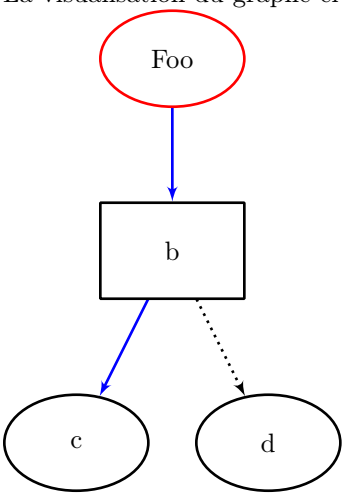
```
digraph mon_graphe {
  // Un attribut appliqué au graphe entier
    size="1,1";

    // Un attribut peut spécifier une étiquette d'un noeud
    a [label="Foo" color=red];

    // Ici l'attribut peut modifier la forme géométrique d'un noeud
    b [shape=box];
```

```
    // Ici les arcs ont des propriétés différentes
    a -> b -> c [color=blue];
    b -> d [style=dotted];
}
```

La visualisation du graphe ci-dessus donnera :




## 4.3 Traduction d'une représentation intermédiaire d'un programme miniC vers le langage DOT

Les déclarations de variables dans un programme miniC ne sont pas reportées dans un arbre abstrait. Les informations sur la déclaration d'une variable serviront pour l'analyse sémantique du compilateur (détection d'erreur par exemple), une telle déclaration n'ajoute ni nœud ni arc dans l'arbre abstrait. Ci-dessous les modèles de traduction d'un programme miniC vers un fichier DOT pour chaque construction syntaxique.

Dans ce qui suit, un triangle en pointillé représente un sous-arbre abstrait quelconque.

### 4.3.1 La déclaration d'une fonction

Chaque fonction dans un programme miniC dont le corps est défini par programme doit être traduite sous forme d'un arbre abstrait, dont la racine est un nœud sous forme de trapèze vers le bas de couleur bleue. L'étiquette du nœud doit contenir le nom de la fonction et son type (soit **int** soit **void**). Par exemple, le programme miniC suivant :

```
void foo (int a, int b){
 ...
}
```

doit être traduit vers l'arbre abstrait de la Figure 1.
Le code DOT d'un tel arbre ressemble à :

```
digraph mon_programme {
    node_mafonction [label="foo, void" shape=invtrapezium color=blue];
    node_reste [shape=triangle label="..." style=dotted];

    node_mafonction ->  node_reste
}
```

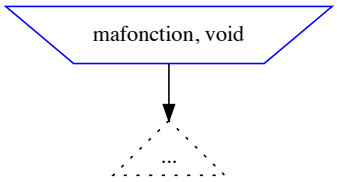



FIGURE 1 – Arbre abstrait d'une fonction

Une fonction déclarée externe n'a pas besoin d'être traduite en un arbre abstrait, car son corps n'est pas défini.

### 4.3.2 Une séquence d'instructions

Une séquence ou un bloc d'instructions `INST1, INST2, INST3, ...` doit être traduite en un arbre dont la racine est appelée BLOC comme montré dans la Figure 2. Chaque instruction du bloc devient ainsi fils de la racine.

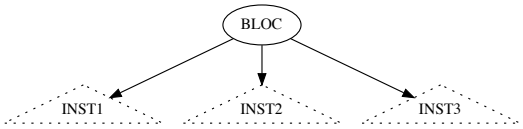



FIGURE 2 – Arbre abstrait d'une séquence d'instructions

Le code DOT d'un tel arbre ressemble à :

```
digraph mon_programme {
   node_bloc [label="BLOC" shape=ellipse];
   // les instructions du bloc doivent êtres mises comme fils de la racine
   node_inst1 [shape=triangle label="INST1" style=dotted];
   node_inst2 [shape=triangle label="INST2" style=dotted];
   node_inst3 [shape=triangle label="INST3" style=dotted];

   node_bloc ->  node_inst1
   node_bloc ->  node_inst2
   node_bloc ->  node_inst3
}
```

### 4.3.3 L'instruction d'affectation

Une instruction d'affectation de la forme `var = EXPR;` doit être traduite en un arbre dont la racine est appelée `:=`. Le fils gauche doit être étiqueté par le nom de la variable affectée, et le fils droit doit être le

sous-arbre décrivant l'expression, comme illustré dans la Figure 3.. Le code DOT d'un tel arbre ressemble

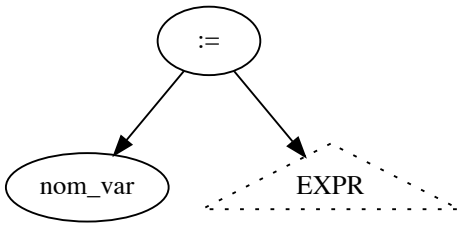



FIGURE 3 – Arbre abstrait d'une affectation de variable

à :

```
digraph mon_programme {
    node_affect [label=":=" shape=ellipse];
    node_var [shape=ellipse label="nom_var"];
    node_expr [shape=triangle label="INST2" style=dotted];

    node_affect -> node_var
    node_affect -> node_expr
}
```

#### 4.3.4 Les expressions arithmétiques ou logiques

Les expressions arithmétiques ou logiques doivent être converties en arbres binaires (qui décrivent le code trois adresses). Chaque nœud de l'arbre binaire est étiqueté par l'opération de base (addition, soustraction, ET logique, OU logique, etc). Le fils gauche contient le premier opérande, et le fils droit le deuxième opérande. Par exemple, le code mini C suivant :

```
toto = x*y - z /t *6;
```

devra être traduit vers l'arbre abstrait de la Figure 4. Le code DOT d'un tel arbre ressemble à :

```
digraph mon_programme {
  node_1 [label=":=" shape=ellipse];
  node_2 [shape=ellipse label="toto"];
  node_3 [shape=ellipse label="-"];
  node_4 [shape=ellipse label="*"];
  node_5 [shape=ellipse label="*"];
  node_6 [shape=ellipse label="/"];
  node_7 [shape=ellipse label="x"];
  node_8 [shape=ellipse label="y"];
  node_9 [shape=ellipse label="z"];
  node_10 [shape=ellipse label="t"];
  node_11 [shape=ellipse label="6"];
```

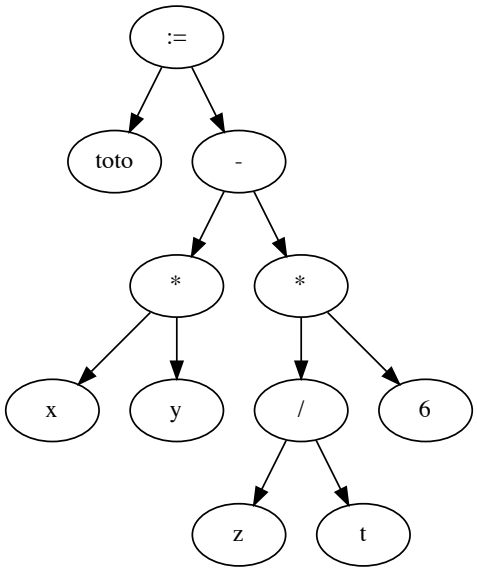



FIGURE 4 – Arbre abstrait d'une affectation de variable avec une expression arithmétique

```
  node_1 -> node_2;
  node_1 -> node_3;
  node_3 -> node_4;
  node_3 -> node_5;
  node_4 -> node_7;
  node_4 -> node_8;
  node_5 -> node_6;
  node_5 -> node_11;
  node_6 -> node_9;
  node_6 -> node_10;
}
```

#### 4.3.5 Les références à un élément de tableau

Une référence à un élément de tableau multidimensionnel comme `montab[i1][i2][i3]` doit être traduite en un arbre binaire dont la racine est étiquetée par TAB. Le premier fils doit contenir le nom du tableau référencé. Le 2e fils contient l'expression du premier indice de tableau (une variable, une constante ou une expression arithmétique). Le 3e fils contient l'expression du 2e indice de tableau, ainsi de suite pour toutes les dimensions du tableau. Ceci est illustré dans la la Figure 5.

#### 4.3.6 Les expressions conditionnelles

Les expressions conditionnelles qui contiennent des opérateurs de comparaison $>$, $<$, $<=$, $>=$, $==$, $!=$ doivent être traduites en arbres binaires. La racine de l'arbre doit être étiquetée par l'opérateur de compa-

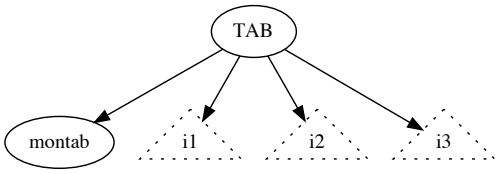



FIGURE 5 – Arbre abstrait d'une référence à un élément de tableau

raison, le fils gauche par la première expression arithmétique, et le fils droit par la deuxième expression arithmétique comme illustré dans la Figure 6.

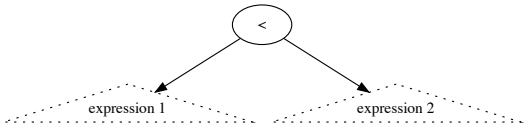



FIGURE 6 – Arbre abstrait d'une expression conditionnelle

#### 4.3.7 L'instruction de retour d'une fonction

Chaque instruction `return` dans une fonction doit être traduite en un nœud sous forme de trapèze vers le haut de couleur bleue, étiqueté avec le mot clé RETURN. Le fils de ce nœud doit correspondre à la valeur retournée. C'est soit une constante, soit une variable simple, soit une expression arithmétique sous forme d'arbre binaire. Par exemple, l'instruction `return toto;` devra être traduite vers le nœud de l'arbre abstrait de la Figure 7. Si l'instruction `return` ne retourne aucune valeur, alors il n'y a pas de nœud fils.

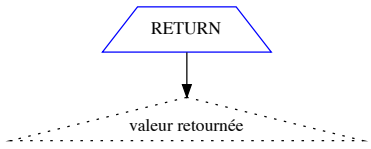



FIGURE 7 – Arbre abstrait d'une instruction de retour de fonction

Le code DOT d'un tel arbre ressemble à :

```
digraph mon_programme {
 node_return [label="RETURN" shape=trapezium color=blue];
 node_exp [shape=invtriangle label="..." style=dotted];
```

```
 node_exp ->   node_return
}
```

#### 4.3.8 L'instruction conditionnelle if-then et if-then-else

Les instructions conditionnelles doivent être traduites en un arbre avec une racine étiquetée avec le mot clé IF, avec une forme de losange. La forme de losange peut être obtenue en utilisant l'attribut `[shape=diamond]` sur le nœud. Le premier fils de la racine est le sous-arbre décrivant la condition sous forme d'arbre binaire. Le deuxième fils est le bloc d'instructions de la branche THEN. Le troisième fils serait éventuellement le bloc des instructions ELSE s'il existe. L'arbre abstrait est illustré dans la Figure 8.

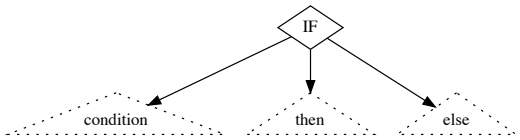



FIGURE 8 – Arbre abstrait d'une instruction conditionnelle

#### 4.3.9 L'instruction switch

Une instruction `switch` doit être traduite en un arbre dont la racine est étiquetée avec le mot clé SWITCH. Le premier fils correspond à l'expression du SWITCH. Ensuite chaque fils correspond au n-ème cas de switch. Le dernier fils doit correspondre au cas par défaut. L'arbre abstrait est illustré dans la Figure 9.

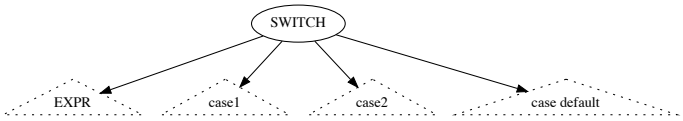



FIGURE 9 – Arbre abstrait d'une instruction switch

#### 4.3.10 L'instruction de boucle for

Une instruction de boucle `for` doit être traduite en un arbre dont la racine est étiquetée avec le mot clé FOR. Le premier fils correspond à l'instruction d'initialisation de la boucle for. Le deuxième fils correspond à la condition du for. Le 3e fils correspond à l'instruction de post-condition. Le 4e fils correspond au corps de la boucle. L'arbre abstrait est illustré dans la Figure 10.

#### 4.3.11 L'instruction de boucle while

Une instruction de boucle `while` doit être traduite en un arbre dont la racine est étiquetée avec le mot clé WHILE. Le premier fils correspond à la condition du while (sous forme d'arbre binaire comme une expression). Le 2e fils correspond au corps de la boucle. L'arbre abstrait est illustré dans la Figure 11.

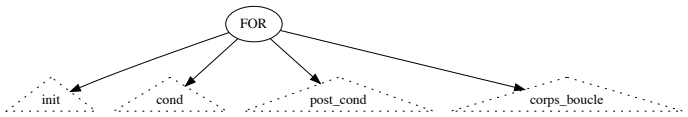



FIGURE 10 – Arbre abstrait d'une instruction for

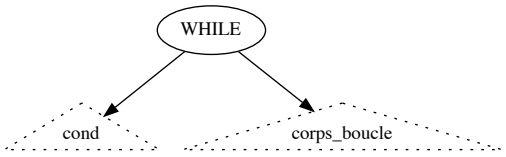



FIGURE 11 – Arbre abstrait d'une instruction while

#### 4.3.12 L'instruction break

L'instruction `break` doit apparaître avec une forme rectangulaire comme illustré dans la Figure 12, et étiquetée avec le mot clé BREAK. La forme rectangulaire peut être obtenue en appliquant l'attribut `shape=box` sur le nœud.

#### 4.3.13 L'appel d'une fonction

Un appel d'une fonction `foo(param1, param2, param3, ...)` doit être traduit en un arbre binaire dont la racine est étiquetée avec le nom de la fonction, avec une forme de polygone. Chaque fils de la racine correspond à un paramètre de l'appel de fonction, comme dans la Figure 13. Le code DOT d'un tel arbre est le suivant :

```
digraph mon_programme {
 node_1 [label="foo" shape=septagon];
 node_2 [shape=triangle label="param1" style=dotted];
 node_3 [shape=triangle label="param2" style=dotted];
 node_4 [shape=triangle label="param3" style=dotted];
 node_1 ->   node_2
 node_1 ->   node_3
 node_1 ->   node_4
}
```

### 4.4 Exemple d'une représentation intermédiaire écrite en DOT

Le code `exempleminiC.c` peut être traduit par exemple en `exempleminiC.dot`. L'exemple code miniC est le suivant :

```
extern int printd( int i );
/*----------------------------*/
```

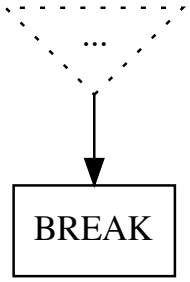



FIGURE 12 – Arbre abstrait d'une instruction break

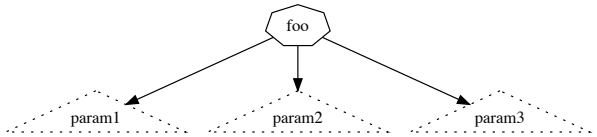



FIGURE 13 – Arbre abstrait d'un appel de fonction

```
/* un petit programme exemple
  pour montrer le langage miniC.*/
/*----------------------------*/
int main() {
   int i, x;
   for (i=0; i<10; i=i+1) {
 if (i==5) break;
   }
   printd(i);
   return (i+3);
}
```

Le compilateur devrait parser et analyser ce code miniC et produire l'arbre abstrait de la Figure 14.

# 5 Compilateur miniC en deux passes

Nous souhaitons que votre compilateur fonctionne en deux passes :

1. La première passe parse le programme miniC en entrée, vérifie les erreurs éventuelles, et construit la

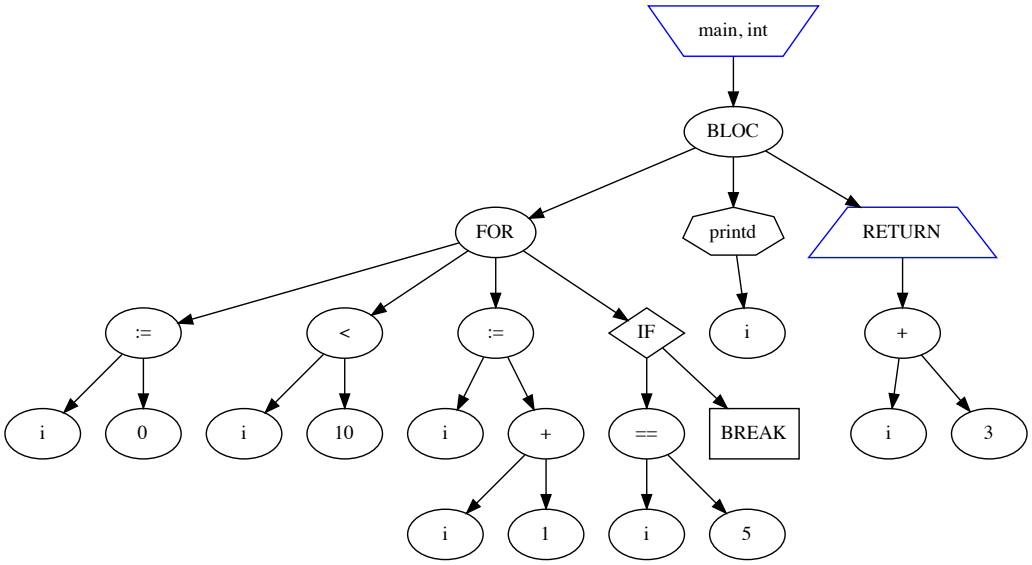



FIGURE 14 – Exemple d'un arbre abstrait d'un programme miniC

représentation intermédiaire **en mémoire**. La représentation intermédiaire est un arbre abstrait pour chaque fonction du programme en entrée.

2. La deuxième passe parcourt la représentation intermédiaire en mémoire, et génère le fichier DOT en sortie si absence d'erreur.

Si le programme en entrée contient une erreur, le compilateur devra afficher un message d'erreur approprié (en indiquant la ligne) et ne générer aucun fichier DOT en sortie. Je vous rappelle qu'un arbre abstrait n'est pas un arbre syntaxique. Un arbre abstrait ne contient aucune information concernant la grammaire du langage en entrée (contrairement à un arbre syntaxique).

# 6 Conseils pratiques

Un projet de compilation est très important dans la vie d'un étudiant en science informatique. Pour beaucoup d'entre vous, c'est probablement la seule occasion dans votre carrière d'écrire un compilateur complet. Il est donc important de faire des efforts pour le réaliser jusqu'au bout, car c'est un travail très formateur que des étudiants des autres disciplines scientifiques ne font pas.

Vous êtes autorisés à communiquer entre vous tous (tous les étudiants), à vous échanger des idées et des astuces, à vous entraider pour comprendre et progresser ensemble. En revanche il est interdit de soumettre un projet qui n'est pas le fruit de votre travail : interdiction formelle de copier le code d'une autre personne, ou qu'une personne fasse le projet à la place d'une autre personne ou même d'utiliser des outils de génération automatique de code ou de rapport. Ceci constituerait une fraude.

Concernant le travail par binôme, nous ne voulons pas qu'un des deux membres fasse tout le projet pour l'autre. Il faut apprendre à travailler en équipe pour apprendre. Les enseignants sont capables de discerner si un étudiant a fourni des efforts ou pas dans un travail de binôme, et peuvent décider de donner des notes

différentes.

Par expérience, les étudiants trouvent un projet de compilation difficile. En vrai, il n'est pas difficile, il est juste compliqué et demande du travail continu. Ci-dessous quelques conseils :

- Le projet de compilation est un effort sur tout le semestre. Commencez le dès que vous abordez l'analyse lexico-syntaxique en TD. N'attendez pas les dernières semaines du semestre pour commencer votre projet, car vous découvrirez des difficultés techniques.
- Commencez par tester votre analyseur lexical en premier lieu, et vérifiez que tous les programmes tests fournis passent correctement (à savoir que les tokens sont correctement analysés par votre analyseur lexical). Vérifiez sa robustesse en modifiant les programmes tests pour provoquer des erreurs lexicales : introduisez des caractères interdits, des noms de variables incorrects, etc.
- Une fois que vous êtes sûrs que votre analyseur lexical fonctionne correctement, vérifiez que votre parseur (analyseur lexico-syntaxique) fonctionne sur tous les programmes tests. Vérifiez sa robustesse en modifiant les programmes tests pour provoquer des erreurs syntaxiques, absence de conflits *shift-reduce* dans le fichier yacc, etc.
- Une fois que vous estimez que votre parseur est fiable, commencez à introduire des routines sémantiques yacc qui affichent des messages appropriés durant la compilation afin de tester que les routines sémantiques s'exécutent correctement. Ce sera le squelette de votre traduction dirigée par la syntaxe, que vous devrez compléter pour la génération de code.
- Attention, en langage C, contrairement à d'autres langages, les chaînes de caractères doivent être allouées en mémoire explicitement. Si une routine sémantique de yacc accède à une chaîne de caractères non allouée en mémoire, votre compilateur plantera. Ensuite, l'usage en programmation est de libérer toutes les zones mémoire allouées dynamiquement.

Je vous souhaite tout le succès.

Pr Sid Touati