# NF11/AI21 : génération de code

Comment un compilateur « fabrique » un programme

---

Stéphane Bonnet

Printemps 2026

Université de Technologie de Compiègne

# Sommaire

Compilateurs

Rappels

Traduction dirigée par la syntaxe

Langage d'illustration : smallC

Analyse sémantique

Code trois adresses

Génération du pré-code machine

Allocation des registres

Génération du code machine final

Conclusion

# Compilateurs

# Qu'est-ce qu'un compilateur ?

- Programme qui traduit un code source (langage haut niveau) en un autre langage (souvent bas niveau)
- Exemple : C → Assembleur / binaire
- Objectif : rendre exécutable un programme sur une machine cible

# Compilation vs Interprétation

**Compilation**

Traduction complète du programme avant exécution

**Interprétation**

Exécution directe ligne par ligne par un interpréteur

# Compilation vs Interprétation

**Compilation**

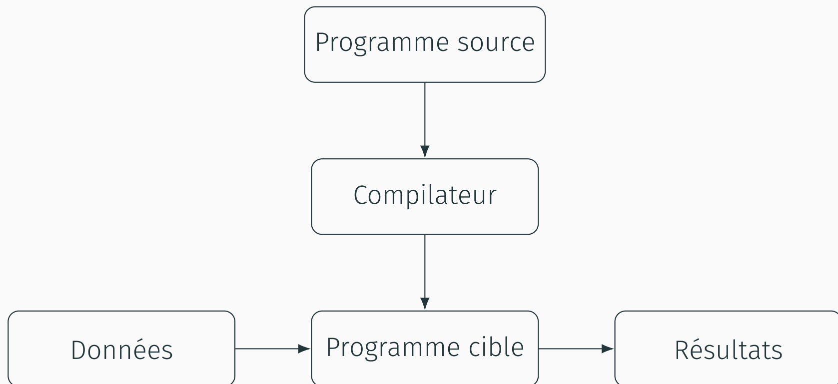
Traduction complète du programme avant exécution

**Interprétation**

Exécution directe ligne par ligne par un interpréteur

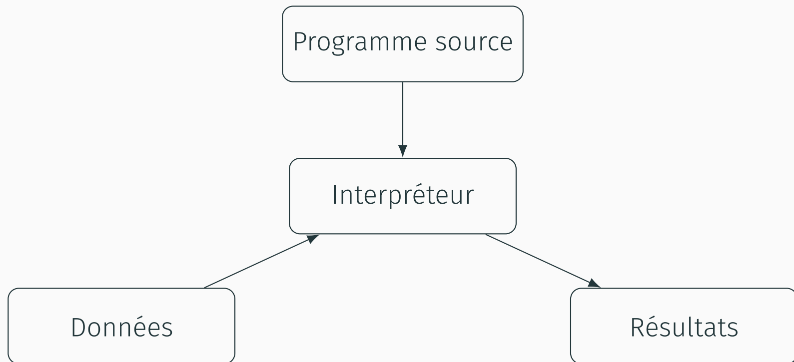
- **Compilateurs** : gcc, rustc, javac, …(vers bytecode)
- **Interpréteurs** : python, php, node.js, …(selon usage)

# Compilateur




Transformation du programme source en un programme exécutable sémantiquement équivalent.

# Interpréteur




Exécution directe des opérations du programme source par l'interpréteur.

# Transpilation

- Transformation d’un code source en un autre code source (même niveau d’abstraction)
- Exemple : TypeScript → JavaScript
- Utilisé pour adapter le code à différentes plateformes ou environnements

# Nécessité d'une analyse syntaxique

- Dans de rares cas, la traduction peut être triviale et se faire mot à mot.
- La plupart du temps : analyser la structure du code source pour le décomposer en éléments significatifs ou constructions du langage source.
- La traduction d'une construction dépend de la position qu'elle occupe dans le programme source.

# Architecture d'un compilateur

Deux grandes étapes :

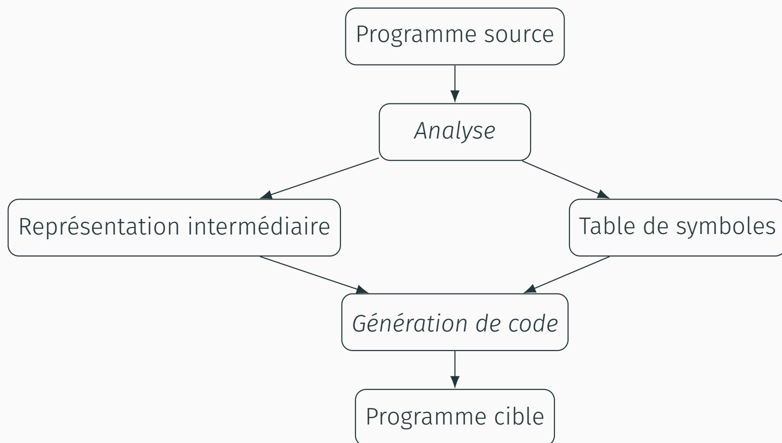
**L'analyse** réalisée par la partie frontale du compilateur (*front-end*) :

- découpe le programme source en ses constituants
- vérifie la syntaxe et la sémantique
- produit une représentation intermédiaire
- conserve dans une table de symboles les informations sur les variables, fonctions, etc.

**La génération de code** réalisée par la partie finale du compilateur (*back-end*) :

- construit le programme cible à partir de la représentation intermédiaire et de la table de symboles

# Architecture d’un compilateur



# Choix de la représentation intermédiaire

C’est un compromis :

- Doit être raisonnablement facile à produire à partir du code source
- Doit être suffisamment riche pour représenter les constructions du langage source
- Doit être raisonnablement facile à transformer en code cible

# Intérêt de la représentation intermédiaire

- Permet de séparer l’analyse syntaxique de la génération de code
- Facilite les optimisations : on peut transformer la représentation intermédiaire avant de générer le code cible
- Permet de cibler plusieurs architectures ou langages cibles à partir d’une même représentation intermédiaire : on obtient $m \times n$ compilateurs en écrivant $m$ analyseurs et $n$ générateurs de code.

# Rappels

# Rappel : analyse syntaxique

- Étape clé de la compilation
- Transforme le texte brut du programme en une structure arborescente
- Utilise une **grammaire formelle** pour définir la syntaxe du langage
- Vérifie la conformité du programme à cette grammaire

En général, deux étapes :

- **Analyse lexicale** : découpe le texte en *tokens* (mots-clés, identificateurs, etc.)
- **Analyse syntaxique** : construit un arbre de dérivation à partir des tokens

# Exemple : expression arithmétique – $(3+5)*8$

## Grammaire simplifiée

$$E \rightarrow E + T \mid T$$

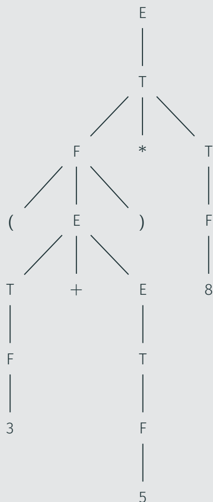
$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid n$$

## Expression

$(3+5)*8$

## Arbre de dérivation



# Traduction dirigée par la syntaxe

# Traduction dirigée par la syntaxe

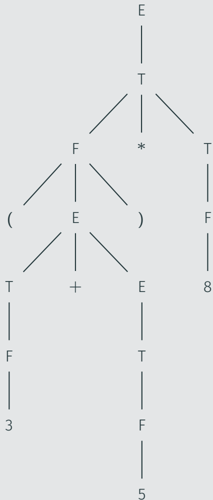
- Utilise la structure syntaxique du programme pour guider la traduction
- Chaque règle de grammaire peut être associée à une action de traduction
- Ces actions sont exécutées :
  - lors de l'analyse syntaxique
  - ou après par parcours de l'arbre de dérivation
- Le résultat de ces exécutions est une traduction du programme analysé
- Elle repose sur deux concepts clés :
  - **Attributs** : informations associées aux nœuds de l'arbre de syntaxe
  - **Actions sémantiques** : opérations à réaliser lors de la construction de l'arbre

# Arbre de syntaxe abstrait (AST, *Abstract Syntax Tree*)

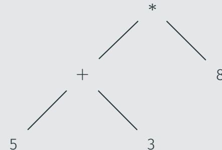
- L'arbre de dérivation est souvent trop détaillé pour la traduction : de nombreux nœuds sont inutiles pour la traduction.
- L'AST permet une analyse sémantique plus naturelle, en ne conservant de l'analyse syntaxique que les éléments pertinents au reste des traitements.
- Il est construit pendant l'analyse syntaxique ou à partir de l'arbre de dérivation grâce à la traduction dirigée par la syntaxe.

# Exemple d'arbre de syntaxe abstrait pour l'expression $(3+5)*8$

## Arbre de dérivation




## Arbre de syntaxe abstrait



# Traduction dirigée par la syntaxe

## Grammaires attribuées

## Grammaire attribuée

Une grammaire avec des actions sémantiques attachées aux règles.

## Action sémantique

Manipule des attributs associés aux nœuds de l'arbre de dérivation.

## Attribut

Information associée à un nœud de l'arbre de dérivation, utilisée pour la traduction.

- Le type d'une expression
- La valeur d'une expression
- Le numéro de ligne d'une déclaration (pour les messages d'erreur)
- Un nœud de l'arbre de syntaxe abstrait
- ...

Notation : $A.t$ pour l'attribut $t$ du nœud $A$.

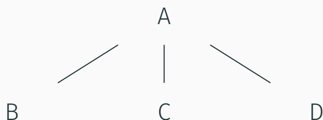
# Attributs hérités et synthétisés

- **Attributs synthétisés** :
  - Calculés à partir des attributs des nœuds enfants
  - Utilisés pour construire des informations à remonter dans l'arbre
- **Attributs hérités** :
  - Passés d'un nœud parent à un nœud enfant
  - Utilisés pour transmettre des informations contextuelles

En pratique, les attributs hérités sont souvent utilisés pour transmettre des informations de contexte (comme le type d'une variable), tandis que les attributs synthétisés sont utilisés pour construire des informations à partir des nœuds enfants (comme le type d'une expression).

# Attributs synthétisés

Un attribut est dit **synthétisé** si sa valeur est calculée à partir des attributs des nœuds enfants et de lui-même.




$$A.t = f(B.t, C.t, D.t)$$

- Ils peuvent être évalués en parcourant l'arbre de dérivation de bas en haut (parcours ascendant).
- Ce parcours peut se faire en même temps que la construction de l'arbre lors de l'analyse syntaxique.
- La grammaire est dite **S-attribuée** si tous les attributs sont synthétisés.

# Attributs hérités

Un attribut est dit **hérité** si sa valeur est calculée à partir des attributs de son père et de lui-même.

- On restreint en général aux attributions
  - synthétisées
  - héritées d'un frère gauche
  - héritées du père (attention aux cycles !)

  On parle alors de grammaire **L-attribuée**.
- Les attributs peuvent être calculés par un parcours en profondeur gauche de l'arbre de dérivation.

# Actions sémantiques

En reprenant l'exemple de la grammaire de l'expression arithmétique, on peut associer des actions sémantiques aux règles :

| Règle | Action |
|---|---|
| $E \rightarrow E + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | $E.val = T.val$ |
| $T \rightarrow T * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | $T.val = F.val$ |
| $F \rightarrow (E)$ | $F.val = E.val$ |
| $F \rightarrow n$ | $F.val = n$ |

Ici tous les attributs sont synthétisés, c'est une grammaire S-attribuée.

# Retour sur l'arbre de syntaxe abstrait

En reprenant l'exemple de l'arbre de syntaxe abstrait pour l'expression $(3 + 5) * 8$, on peut voir comment les actions sémantiques permettent de construire cet arbre.

**nœud(op, g, d)**

Un nœud de l'arbre de syntaxe abstrait est construit par l'action sémantique associée à la règle de grammaire :

- *op* est l'opérateur
- *g* est le sous-arbre gauche
- *d* est le sous-arbre droit

# Retour sur l'arbre de syntaxe abstrait

## Actions sémantiques associées

| Règle | Action |
| --- | --- |
| $E \to E + T$ | $E.n = \text{nœud}(+, E_1.n, T.n)$ |
| $E \to T$ | $E.n = T.n$ |
| $T \to T * F$ | $T.n = \text{nœud}(*, T_1.n, F.n)$ |
| $T \to F$ | $T.n = F.n$ |
| $F \to (E)$ | $F.n = E.n$ |
| $F \to n$ | $F.n = n$ |

# Langage d'illustration : smallC

# Le langage smallC

Pour illustrer la génération de code, nous allons utiliser un langage de programmation simple, smallC.

- Inspiré du langage C, mais très simplifié
- Construit autour de la syntaxe des expressions arithmétiques et des instructions de contrôle.

# Exemples de programmes en smallC

- Calcul de la somme de deux entiers :

```
a = 3;
b = 5;
c = a + b;
return c;
```

- Boucle de 1 à 10 :

```
i = 1;
while (i <= 10) {
    i = i + 1;
}
return i;
```

- Expression conditionnelle :

```
if (a > b) {
    a = a - b;
} else {
    b = b - a;
}
return a;
```

# Grammaire de smallC

```
<program> ::=  <stmt_list> | ""
<stmt_list> ::= <stmt> <stmt_list> | <stmt>
<stmt> ::= <assign_stmt> | <if_stmt> | <while_stmt> | <return_stmt>
<assign_stmt> ::= <ident> "=" <eq_expr> ";"
<if_stmt> ::= "if" "(" <eq_expr> ")" "{" <stmt_list> "}"
            | "if" "(" <eq_expr> ")" "{" <stmt_list> "}" "else" "{" <stmt_list> "}"
<while_stmt> ::= "while" "(" <eq_expr> ")" "{" <stmt_list> "}"
<return_stmt> ::= "return" <eq_expr> ";"
<eq_expr> ::= <rel_expr> | <eq_expr> "==" <rel_expr> | <eq_expr> "!=" <rel_expr>
<rel_expr> ::= <add_expr>
             | <rel_expr> "<" <add_expr>
             | <rel_expr> "<=" <add_expr>
             | <rel_expr> ">" <add_expr>
             | <rel_expr> ">=" <add_expr>
<add_expr> ::= <mul_expr>
             | <add_expr> "+" <mul_expr>
             | <add_expr> "-" <mul_expr>
<mul_expr> ::= <primary_expr>
             | <mul_expr> "*" <primary_expr>
             | <mul_expr> "/" <primary_expr>
<primary_expr> ::= <identifier> | <constant> | <string> | "(" <eq_expr> ")"
```

# Un AST pour smallC

Un programme en smallC est essentiellement composé d'expressions et d'instructions :

- Expressions arithmétiques : addition, soustraction, multiplication, division
- Expressions relationnelles : égalité, inégalité, comparaison
- Instructions de contrôle : `if...else`, `while`, `return`
- Instructions d'affectation : `a = b`, `c = a + b`

On peut définir un nœud adapté à chaque type d'expression ou d'instruction, et construire un arbre de syntaxe abstraite (AST) à partir du programme source.

# Nœuds d'expression

Par exemple, pour les expressions arithmétiques et logiques, les nœuds seraient de la forme :

- `<(left, right)` pour l'addition
- `-(left, right)` pour la soustraction
- `*(left, right)` pour la multiplication
- `/(left, right)` pour la division
- `==(left, right)` pour l'égalité
- `>(left, right)` pour la comparaison
- etc.

# Nœuds d'instruction

On peut spécifier des nœuds différents pour chaque type d'instruction ou préférer un nœud générique `Instruction` avec un champ `type` pour distinguer les types d'instructions.

C'est un choix de conception qui dépend de la complexité du langage et des besoins du compilateur. C'est le choix que nous ferons ici.

- `Affectation(expr)` pour les instructions d'affectation
- `If(condition, then_block, else_block)` pour les instructions conditionnelles
- `While(condition, block)` pour les boucles
- `Return(value)` pour les instructions de retour

# Nœud générique d'instruction

Ce qui conduit à un nœud générique avec les champs suivants :

- `type` : le type de l'instruction (affectation, conditionnelle, boucle, etc.)
- `expr` : pour les conditions des instructions d'affectation, les alternatives, les boucles
- `body` : pour les instructions de bloc (conditionnelles, boucles)
- `else` : pour les instructions conditionnelles
- `next` : prochaine instruction dans l'arbre.

# Construction de l’AST à partir de la grammaire attribuée

La grammaire attribuée de smallC permet de construire l’AST en associant des actions aux règles de la grammaire. Par exemple :

| Règle | Action sémantique |
| --- | --- |
| `IS` $\rightarrow$ `if (E) {SL}` | `IS.s = create_stmt(IF_ELSE, E.e, SL1.s, 0, 0)` |
| `IS` $\rightarrow$ `if (E) { SL } else { SL }` | `IS.s = create_stmt(IF_ELSE, E.e, SL1.s, SL2.s, 0)` |
| `WS` $\rightarrow$ `while (E) { SL }` | `WS.s = create_stmt(WHILE, E.e, SL.s, 0, 0)` |

# Un exemple

Voici un exemple de programme en smallC, suivi de son AST :

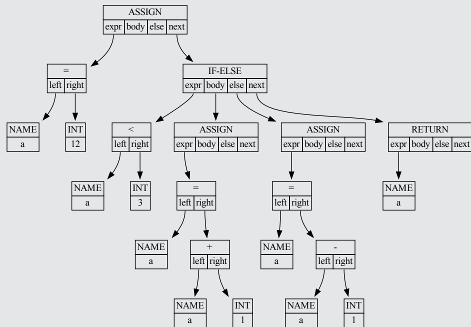
## Programme smallC

```
a = 12;

if (a < 3) {
  a = a + 1;
} else {
  a = a - 1;
}

return a;
```

## AST



# Analyse sémantique

# Analyse sémantique

C'est la phase de la compilation qui vérifie la validité sémantique du programme. un programme peut être syntaxiquement correct mais sémantiquement incorrect :

- Variables non déclarées
- Fonctions avec un nombre incorrect d'arguments
- Opérateurs appliqués à des types incompatibles
- Pas de return dans une fonction non void et vice-versa
- etc.

# Exemple : analyse sémantique

Certaines contraintes ne peuvent pas être vérifiées par une grammaire hors-contexte. Par exemple, comment vérifier que les variables sont déclarées avant d'être utilisées ?

- On utilise une **table de symboles** pour stocker les informations sur les variables et fonctions
- Lors de l'analyse syntaxique, on remplit cette table en parcourant l'arbre de syntaxe abstraite
- Lors de l'analyse sémantique, on vérifie que les variables sont déclarées avant d'être utilisées : une variable doit être présente dans la table de symboles avant son utilisation dans une expression ou une instruction.
- On peut aussi vérifier les types des expressions, la cohérence des déclarations de fonctions, etc.

# Table de symboles

La table de symboles est une structure de données qui stocke les informations sur les variables, fonctions, types, etc. du programme :

- Pour chaque variable : nom, type, portée, position mémoire, taille en mémoire
- Pour chaque fonction : nom, type de retour, paramètres, portée

Elle est généralement implémentée comme un dictionnaire ou une table de hachage.

**Attention**

Il peut y a voir plusieurs tables de symboles, par exemple une pour chaque portée (globale, fonction, bloc, etc.). Les tables locales peuvent être créées et détruites au fur et à mesure de l'analyse du programme ou être conservées.

# Exemple de table de symboles

| Nom | Type | Portée | Position mémoire | Taille |
|---|---|---|---|---|
| a | int | globale | 0x1000 | 4 |
| b | int | globale | 0x1004 | 4 |
| c | int | locale (main) | 0x2000 | 4 |
| f | function(int a, int b) $\rightarrow$ int | globale | 0x3000 | 8 |
| a | int | locale (f) | 0x2004 | 4 |
| b | int | locale (f) | 0x2008 | 4 |

# Code trois adresses

# Représentations intermédiaires

Les représentations intermédiaires ne sont pas forcément des arbres de syntaxe abstraite. Elles peuvent être :

- Des graphes de flot de contrôle (CFG) pour représenter les instructions et les sauts
- Des listes d'instructions pour représenter les opérations de base
- Des représentations intermédiaires spécifiques à un compilateur, comme le code à trois adresses (TAC).

**Attention**

Il est possible d'utiliser plusieurs représentations intermédiaires au cours de la compilation, pour faciliter les transformations et optimisations.

## Représentations intermédiaires

- Arbre abstrait : représentation « haut niveau », structurée et proche du langage source
- Code trois adresses (TAC) : représentation « bas niveau », proche du code machine, linéaire, peu structurée

La génération du code trois adresses se fait en parcourant l'arbre abstrait.

# Exemple de TAC pour smallC

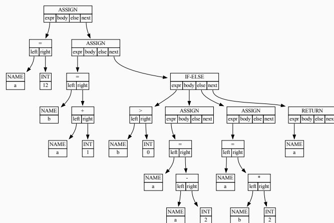
Code source en smallC

```
a = 12;
b = a + 1;
if (b > 0) {
    a = a - 2;
} else {
    a = b * 2;
}

return a;
```

Code à trois adresses (TAC)

```
00    a = 12
01    t0 = a + 1
02    b = t0
03    t1 = b > 0
04    if (t1 == 0) goto L0
05    t2 = a - 2
06    a = t2
07    goto L1
    L0:
08    t3 = b * 2
09    a = t3
    L1:
10    return a
```



# Pourquoi le TAC ?

- Se concentrer sur le linéarisation du programme
    - dérouler les boucles
    - désimbriquer les blocs de code
    - décomposer les expressions complexes en séquences d'opérations simples
- Faire abstraction des détails de la machine cible
    - Organisation de la mémoire
    - Nombre de registres disponibles
    - Instructions spécifiques
    - etc.
- Facile à convertir en code machine mais assez générique pour s'adapter à différentes architectures. C'est un compromis.

## Représentation du TAC

- Il s’agit d’une séquence d’instructions, chacune ayant un opérateur et des opérandes.
- L’indice de l’opération dans la séquence est son **adresse**.
- Chaque instruction peut être représentée par un tuple de la forme :

```
(op, a1, a2, r)
```

- Les opérandes peuvent être des variables, des constantes ou des adresses mémoire.
  - `a1` et `a2` sont en général les opérandes de l’opération
  - `result` est en général le résultat de l’opération
- `op` est une opération simple : arithmétique, logique, sauts, etc.

# Les opérandes

- Les opérandes peuvent être :
  - des constantes (entiers, flottants, etc.)
  - des variables (noms de variables du programme)
  - des étiquettes du code TAC (pour les sauts)
  - des variables temporaires générées
- Les constantes et variables existent dans le programme source (et donc dans l’AST).
- Les étiquettes et variables temporaires sont créées lors de la génération du TAC.

# Variables temporaires

- Les expressions complexes du langage source ne sont pas autorisées dans le TAC : au plus trois opérandes par instruction.
- Il faut les décomposer et stocker les valeurs intermédiaires dans des **variables temporaires** :

```
delta = b * b - 4 * a * c;
```

```
00  t2 = 4 * a
01  t1 = t2 * c
02  t3 = b * b
03  t0 = t3 - t1
04  delta = t0
```

- Elles sont générées à la demande pendant la traduction
- Elles sont nommées de manière unique, `t0`, `t1`, etc. par convention

# Étiquettes

- Une étiquette est un nom symbolique donné à une adresse
- Elles permettent de référencer des instructions spécifiques dans le code TAC.

```
À la place des adresses numériques...
 00    t0 = a
 01    if (t0 == 0) goto 05
 02    t2 = 3 * a
 03    t1 = t2 + 1
 04    return t1
 05    return a
```

```
... on utilise des étiquettes.
 00    t0 = a
 01    if (t0 == 0) goto L0
 02    t2 = 3 * a
 03    t1 = t2 + 1
 04    return t1
     L0:
 05    return a
```

# Les variables

- Ce sont des noms du langage source.
- Dans le TAC, on a besoin d'informations sur ces variables contenues dans la table des symboles
- Les variables sont représentées par une référence vers une entrée de la table des symboles.

# Les instructions

- Les instructions décrivent les opérations et leurs opérandes.
- Par exemple :

| Type | Opérations | a1 | a2 | r | Syntaxe |
|---|---|---|---|---|---|
| Arithmétique | + - * / | ctv | ctv | tv | `r = a1 op a2` |
| Affectation | = | ctv | | tv | `r = a1` |
| Saut test | == != < > <= >= | ctv | ctv | e | `if a1 op a2 goto e` |
| Saut direct | goto | e | | | `goto e` |
| Return | return | ctv | | | `return a1` |

- Chaque opération accepte un certain nombre de types d'opérandes : constantes(c), variables(v), temporaires(t), étiquettes(e).

# Traduction

- La traduction de l'AST en TAC se fait lors d'un parcours en profondeur de l'arbre.
- On peut utiliser à nouveau des grammaires attribuées fondées sur :
    - Une grammaire correspondant à l'arbre abstrait
    - Des attributs synthétisés pour les variables intermédiaires
    - une fonction `gen()` pour générer une ligne de code TAC
    - Des fonctions auxiliaires pour générer des temporaires et des étiquettes

# Traduction des expressions

- Principe : à l'issue de l'exécution du code correspondant à une expression, le résultat est dans une variable ou un temporaire.
  - Pour une constante, elle est renvoyée telle quelle
  - Pour une variable, le nom de la variable est renvoyé
  - Pour une opération, l'opération est décomposée et le résultat est renvoyé dans un temporaire.

# Exemples d’actions sémantiques pour les expressions

| Règle | Action sémantique |
|---|---|
| $E \rightarrow E_1 + E_2$ | $E.t = gen_temp(); gen(E.t = E_1.t + E_2.t);$ |
| $E \rightarrow E_1 - E_2$ | $E.t = gen_temp(); gen(E.t = E_1.t - E_2.t);$ |
| ... | ... |
| $E \rightarrow n$ | $E.t = n.val$ |
| $E \rightarrow v$ | $E.t = v.name$ |

# Exemples d'actions sémantiques pour les instructions de contrôle

- Production $S \to \text{if}\ (E)\ SL_1\ \text{else}\ SL_2$ :

$E.code$
$e_1 = gen_label()$
$e_2 = gen_label()$
$gen(\text{if } E.t = 0 \text{ goto } e_1)$
$SL_1.code$
$gen(\text{ goto } e_2)$
$gen(e_1)$
$SL_2.code$
$gen(e_2)$

```
E.code
if (E.t == 0) goto e1
SL1.code
goto e2
e1:
SL2.code
e2:
```

# Génération du pré-code machine

# Génération du pré-code machine

- Ce n’est pas encore le code final.
- On considère qu’on dispose d’une infinité de registres `r0`, `r1`, `r2`, etc.
- L’affectation aux registres réels se fera plus tard, lors de l’allocation des registres.
- Tout le reste est identique au code machine final.

Code trois adresses

```
00    t0 = a
01    if (t0 == 0) goto L0
02    t2 = 3 * a
03    t1 = t2 + 1
    L0:
04    return t1
```

Pré-code machine

```
    mov r0, dword ptr[a]
    cmp r0, 0
    je L0
    mov r2, 3
    imul r2, dword ptr[a]
    mov r1, r2
    add r1, 1
L0: mov eax, r1
```

# Idée générale de la génération du pré-code machine

- On parcourt le code TAC ligne par ligne.
- Pour chaque instruction, on génère une ou plusieurs instructions machine.
- Les opérandes du code trois adresses deviennent des registres, des adresses mémoire ou des constantes.

# Exemple de génération pour les opérations arithmétiques

Code trois adresses

```
00    t0 = t1 + 3
```

Pré-code machine

```
mov r0, r1
add r0, 3
```

Le code généré dépend de l'opération et du type des opérandes. Les temporaires `t0`, `t1` sont remplacés par des registres, et les constantes sont directement intégrées dans les instructions.

# Exemple de génération pour les sauts

Code trois adresses

```
00    if t1 == 0 goto L0
```

Pré-code machine

```
cmp r1, 0
je L0
```

Toutes les instructions du TAC sont traduites en suites d'instructions machine équivalentes. Ici :

- L'instruction `cmp destination, source` calcule destination − source et met à jour les indicateurs de condition.
- Si destination = source, `ZF` (Zero Flag) est mis à 1.
- L'instruction `je adr` effectue un saut si `ZF` est à 1 (va à l'étiquette `adr` si la condition est vraie).

# Allocation des registres

# Allocation des registres

- Le nombre de registres disponibles dans une machine est limité. Par exemple sur un processeur x86, il y a 16 registres généraux, mais seulement 8 sont utilisables pour les opérations arithmétiques et logiques.
  - `rax`, `rbx`, `rcx`, ...,`r15`
- Le code TAC peut contenir un nombre illimité de temporaires, mais il faut les associer aux registres disponibles.
- Deux temporaires `t1` et `t2` peuvent être mis dans le même registre s'ils ne sont pas utilisés en même temps.
- L'analyse du TAC ou du pré-code machine permet de déterminer quels temporaires sont actifs à chaque point du programme.

# Allocation des registres

## Analyse de vie des temporaires

# Vie des temporaires

- Un temporaire est vivant s'il contient une valeur qui sera utilisée plus tard dans le programme.
- L'analyse de vie des temporaires permet d'identifier pour chaque instruction *i* du TAC quels temporaires sont en vie au moment de l'exécution de *i*.
- On peut faire cette analyse également en utilisant le pré-code machine.

# Graphe d'analyse

- Le programme est représenté sous forme d'un graphe orienté.
- Chaque instruction du TAC est un sommet.
- Si $I_0$ peut être suivie de $I_1$, il y a un arc de $I_0$ vers $I_1$.

# Exemple

## Code smallC

```
n = 10;
i = 0;
while (i < (n - 1)) {
    i = i + 1;
}
return i;
```

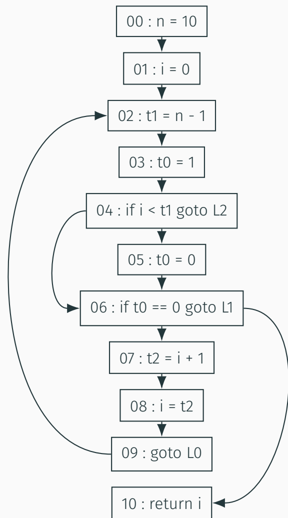
## code TAC

```
00    n = 10
01    i = 0
    L0:
02    t1 = n - 1
03    t0 = 1
04    if i < t1 goto L2
05    t0 = 0
    L2:
06    if t0 == 0 goto L1
07    t2 = i + 1
08    i = t2
09    goto L0
    L1:
10    return i
```

# Graphe d’analyse de vie des temporaires

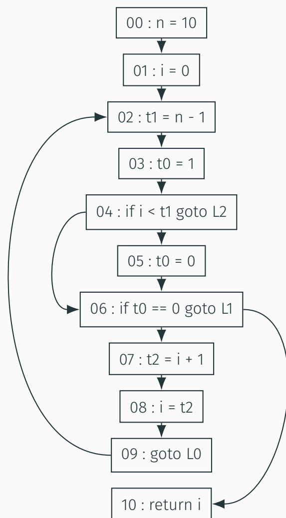
code TAC

```
00    n = 10
01    i = 0
    L0:
02    t1 = n - 1
03    t0 = 1
04    if i < t1 goto L2
05    t0 = 0
    L2:
06    if t0 == 0 goto L1
07    t2 = i + 1
08    i = t2
09    goto L0
    L1:
10    return i
```



# Exemple

- `t2` est utilisée en 8 et définie en 7, donc vivant sur $7 \to 8$.
- `t0` est utilisée en 5 et définie en 5, donc vivant sur $5 \to 6$.
- `t0` est définie en 3 et il existe $3 \to 4$, et $4 \to 6$, donc vivant sur ces deux arcs.
- `t1` est utilisée en 4 et définie en 2, donc vivant sur $2 \to 3$ et $3 \to 4$.



# Résultats de l'analyse

| Arc | t0 | t1 | t2 |
|---|---|---|---|
| $0 \to 1$ | | | |
| $1 \to 2$ | | | |
| $2 \to 3$ | | X | |
| $3 \to 4$ | X | X | |
| $4 \to 5$ | X | | |
| $4 \to 6$ | X | | |
| $5 \to 6$ | | | |
| $6 \to 7$ | | | |
| $6 \to 10$ | X | | |
| $7 \to 8$ | | | X |
| $8 \to 9$ | | | |
| $9 \to 2$ | | | |
| $9 \to 10$ | | | |

Conclusions

- `t0` et `t2` ne sont jamais en vie sur un même arc.
- `t1` et `t2` ne sont jamais en vie sur un même arc.
- `t0` et `t1` sont en vie sur les arcs $3 \to 4$, impossible de les mettre dans le même registre.
- On peut mettre `t0` et `t2` dans le même registre, ainsi que `t1` et `t2` : deux registres suffisent.

# Définition et utilisation des variables

- une affectation à une variable la **définit**.
- $def(s)$ est l'ensemble des variables définies par le sommet $s$.
- Une variable qui apparaît dans une expression ou une instruction est **utilisée**.
- $use(s)$ est l'ensemble des variables utilisées par le sommet $s$.

# Calcul de la vie des variables

- Une variable est **vivante** sur un arc s'il existe un chemin depuis cet arc vers une utilisation de la variable qui ne passe pas par une défintion.
- Une variable est vivante en entrée d'un sommet $s$ si elle est vivante sur au moins un arc entrant de $s$.
- Une variable est vivante en sortie d'un sommet $s$ si elle est vivante sur au moins un arc sortant de $s$.
- On peut associer à chaque sommet $s$ un ensemble de variables vivantes en entrée $in(s)$ et en sortie $out(s)$.
- L'objectif est de calculer ces ensembles pour chaque sommet du graphe d'analyse de vie des temporaires.

# Algorithme de calcul de la vie des variables

**Algorithm 1** Calcul de la vie des variables

1 : Initialiser $in(s) = \emptyset$ et $out(s) = def(s)$ pour chaque sommet $s$.
2 : **repeat**
3 : Pour chaque sommet $s$ :
4 : $in(s) = use(s) \cup (out(s) \setminus def(s))$
5 : $out(s) = \bigcup_{s' \in succ(s)} in(s')$
6 : **until** convergence
7 : Retourner les ensembles $in(s)$ et $out(s)$ pour chaque sommet $s$.

# Allocation des registres

Allocation des registres

# Allocation des registres

- L'allocation des registres consiste à associer les temporaires aux registres disponibles.
- On utilise l'analyse de vie des temporaires pour déterminer quels temporaires peuvent être mis dans le même registre.
- On peut utiliser un algorithme de coloration de graphe pour allouer les registres.
- Chaque registre est une couleur, et chaque temporaire est un sommet du graphe.
- Deux temporaires qui sont vivants en même temps ne peuvent pas être associés au même registre (couleur).

# Graphe d’interférence

- Le graphe d’interférence est un graphe non orienté où :
    - chaque sommet représente un temporaire,
    - il y a une arête entre deux sommets si les temporaires correspondants sont vivants en même temps.
- L’allocation des registres revient à colorier ce graphe avec le nombre de couleurs égal au nombre de registres disponibles.
- Si le graphe est colorable avec le nombre de registres disponibles, l’allocation est possible.

# Construction du graphe d'interférence

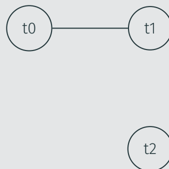
**Algorithm 2** Construction du graphe d'interférence

1 : Initialiser un graphe vide $G$.
2 : **for** chaque sommet $s$ du graphe d'analyse **do**
3 :     **for** chaque temporaire $t \in in(s) \cup out(s), t \notin G$ **do**
4 :         Ajouter le sommet $t$ dans $G$.
5 :     **end for**
6 :     **for** $(t, t') \in in(s) \times in(s), t \neq t'$ **do**
7 :         Ajouter une arête entre $t$ et $t'$ dans $G$.
8 :     **end for**
9 :     **for** $(t, t') \in out(s) \times out(s), t \neq t'$ **do**
10 :         Ajouter une arête entre $t$ et $t'$ dans $G$.
11 :     **end for**
12 : **end for**
13 : **return** le graphe d'interférence $G$.

# Exemple de graphe d'interférence

| s | in(s) | out(s) |
|---|---|---|
| 0 | $\emptyset$ | $\emptyset$ |
| 1 | $\emptyset$ | $\emptyset$ |
| 2 | $\emptyset$ | $\{t1\}$ |
| 3 | $\{t1\}$ | $\{t0, t1\}$ |
| 4 | $\{t0, t1\}$ | $\{t0\}$ |
| 5 | $\emptyset$ | $\{t0\}$ |
| 6 | $\{t0\}$ | $\emptyset$ |
| 7 | $\emptyset$ | $\{t2\}$ |
| 8 | $\{t2\}$ | $\emptyset$ |
| 9 | $\emptyset$ | $\emptyset$ |
| 10 | $\emptyset$ | $\emptyset$ |

## Graphe d'interférence



# Coloration du graphe d'interférence

On va maintenant colorer le graphe d'interférence en utilisant un algorithme de coloration de graphe avec $K$ couleurs au plus.

## Problème de coloration de graphe

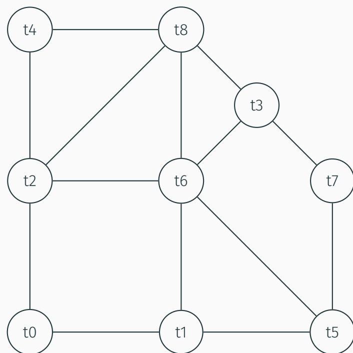
- On cherche à colorer les sommets en utilisant le moins de couleurs possible.
- Deux sommets adjacents ne peuvent pas avoir la même couleur.
- Algorithme NP-complet : il n'existe pas d'algorithme polynomial pour le résoudre en général, mais on peut utiliser des heuristiques.

- Supposons que $G$ possède un sommet $s$ de degré inférieur à $K$.
- $G' = G \setminus s$ est le graphe obtenu en supprimant $s$ de $G$.
- Si on peut colorer $G'$ avec $K$ couleurs, on peut colorer $G$ en attribuant à $s$ une couleur différente de celle de ses voisins : les voisins de $s$ ont au plus $K - 1$ couleurs différentes, donc il reste au moins une couleur disponible pour $s$.

# Coloration du graphe d’interférence (suite)

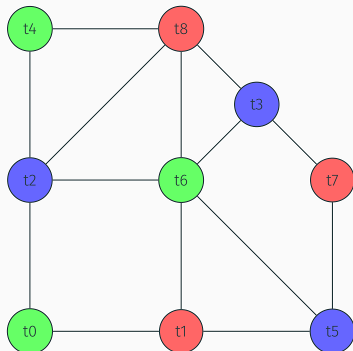
- Algorithme récursif : on enlève un sommet de degré inférieur à $K$, on l’empile dans la liste des sommets colorés et on applique la coloration sur le graphe réduit.
- Idée : l’élimination de sommets de degré inférieur à $K$ permet de réduire le graphe et de faciliter la coloration.
- Si on arrive à un graphe vide, on a réussi à colorer le graphe d’interférence.

# Exemple sur un graphe un peu plus complexe




Essayons de colorer ce graphe avec $K = 4$ couleurs : rouge, vert, bleu et jaune.

# Résultat




On a réussi à colorer le graphe avec 3 couleurs.

# Allocation des registres

- On associe chaque couleur à un registre réel.
- On remplace les temporaires par les registres correspondants dans le code TAC ou le pré-code machine.
- Si un temporaire n’a pas pu être associé à un registre, on le stocke dans la pile ou dans la mémoire (débordement, *spilling*).
- Remarques :
    - Le compilateur peut pré-colorer certains temporaires dans des registres spécifiques pendant la construction du TAC (par exemple, les registres de retour de fonction). Dans ce cas, on ne les colore pas.

# Génération du code machine final

# Génération du code machine final

- Reprendre le pré-code machine
- Remplacer les temporaires par les registres réels qui leurs sont alloués.
- Assembler le code pour le traduire en instructions binaires exécutables par la machine cible.

Et voilà, on a un programme exécutable !

# Conclusion

# Optimisations

Le processus de compilation peut inclure de nombreuses optimisations pour améliorer la performance du code généré :

- Optimisations au niveau de l'arbre abstrait (AST) :
  - Suppression des nœuds inutiles (codes morts)
  - Simplification des expressions
  - Propagation des constantes
  - Élimination des redondances
- Optimisations au niveau du code trois adresses :
  - Réduction du nombre d'instructions
  - Réutilisation des temporaires
- Optimisations au niveau du code machine (*peephole optimizations*) :
  - Réorganisation des instructions pour minimiser les accès mémoire
  - Utilisation efficace des registres

# Diversité des représentations intermédiaires

Il existe de nombreuses RI, linéaires ou arborescentes, adaptées à différents besoins et contextes de compilation. Voici quelques exemples :

- Graphes de flot de contrôle (CFG)
- Graphes de dépendance
- Représentations intermédiaires spécifiques (comme le code à trois adresses)
- Représentations en SSA (Static Single Assignment), utilisée par exemple dans LLVM ou GCC, etc.

# Conclusion

il existe de nombreuses approches pour la compilation et la génération de code, chacune adaptée à des besoins spécifiques. Ce cours a présenté les concepts fondamentaux de la compilation, en proposant une approche moderne, mais ce n'est pas la seule possible : on peut se passer de l'AST et n'utiliser que des représentations intermédiaires linéaires, ou au contraire générer le code machine directement à partir de l'AST.

# Références

# Références


1. Aho, A., Lam, S., Sethi, R., Ullman, J. D. *Compilers : Principles, Techniques, and Tools*. 2nd edition. Person Education, 2007.
2. Appel, W. *Modern Compiler Implementation in C*. 3rd edition. Cambridge University Press, 2004.
3. Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
4. Cooper, K. N., Torczon, L. *Engineering a Compiler*. 3rd edition. Morgan Kaufmann, 2022.
5. Thain, D. *Introduction to Compiler Design. Second Edition*. University of Notre Dame, 2023. `http://compilerbook.org/`