UNIVERSITÉ DE STRASBOURG

(Ré)introduction à la compilation

Cédric Bastoul

cedric.bastoul@unistra.fr

Université de Strasbourg

Compilateur

Définition [Wikipédia]

Un compilateur est un programme informatique qui transforme un code source écrit dans un langage (le langage source) en un autre langage (le langage cible)

- Traducteur d'un code vers un autre
  - Humains et machines n'utilisent pas les mêmes langages
  - Différents langages et niveaux de langages coexistent
  - Source et cible peuvent se confondre (pour optimiser)
- Rapporteur des erreurs contenues dans le code source
  - Identificateurs mal formés
  - Constructions syntaxiques incorrectes
  - Expressions mal typées, etc.

Objectifs de la présentation

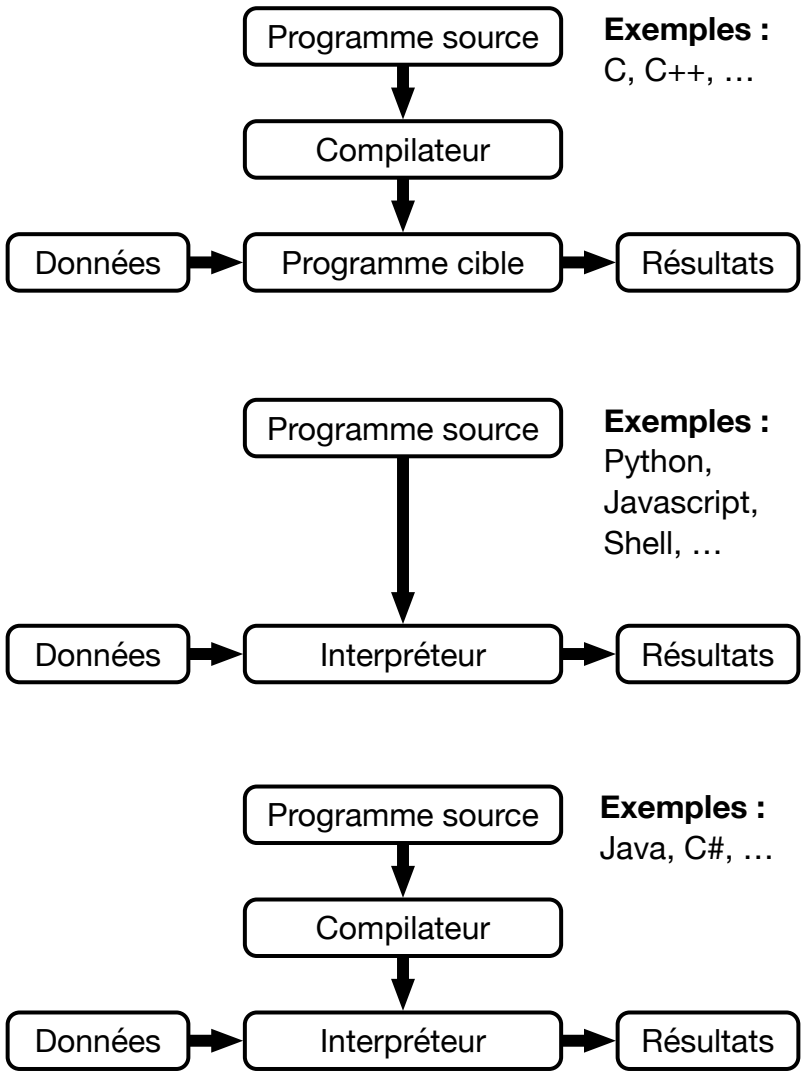
- (Re)découvrir l'intérêt de l'étude de la compilation
- (Re)comprendre le fonctionnement et la construction des compilateurs
- (Re)prendre connaissance des outils fondamentaux pour le travail sur fichiers formatés
- (Re)visiter les aspects ouverts au niveau recherche

Niveaux de langages

- Langages spécifiques (ex. : Matlab, Halide, Shell scripts...)
  - Langages spécialisés disposant de constructions (données et/ou contrôle) spécifiques à un domaine en particulier
- Langage de haut niveau (ex. : Java, C#, C/C++...)
  - Langages généralistes adaptés aux algorithmes et données évolués indépendamment de l'architecture cible
- Langages intermédiaires (ex. : GIMPLE, Bytecode Java...)
  - Langages internes aux compilateurs, communs à tous les langages supportés et adaptés à leur optimisation
- Langages d'assemblage (ex. : assembleur x86 ou ARM...)
  - Langage proche du langage machine mais où instructions et adresses sont remplacées par des noms lisibles
- Langages machine (ex. : x86, ARM...)
  - Suite binaire directement interprétable par l'architecture

Formes de traduction

- **Compilation** : traduction d'un programme en un autre tel que pour toutes données en entrée, le résultat des deux programmes est le même
- **Interprétation** : utilisation d'un programme qui, étant donné un programme source et des données, calcule le résultat du programme source
- **Machine virtuelle** : compilation vers un langage intermédiaire puis interprétation de ce langage

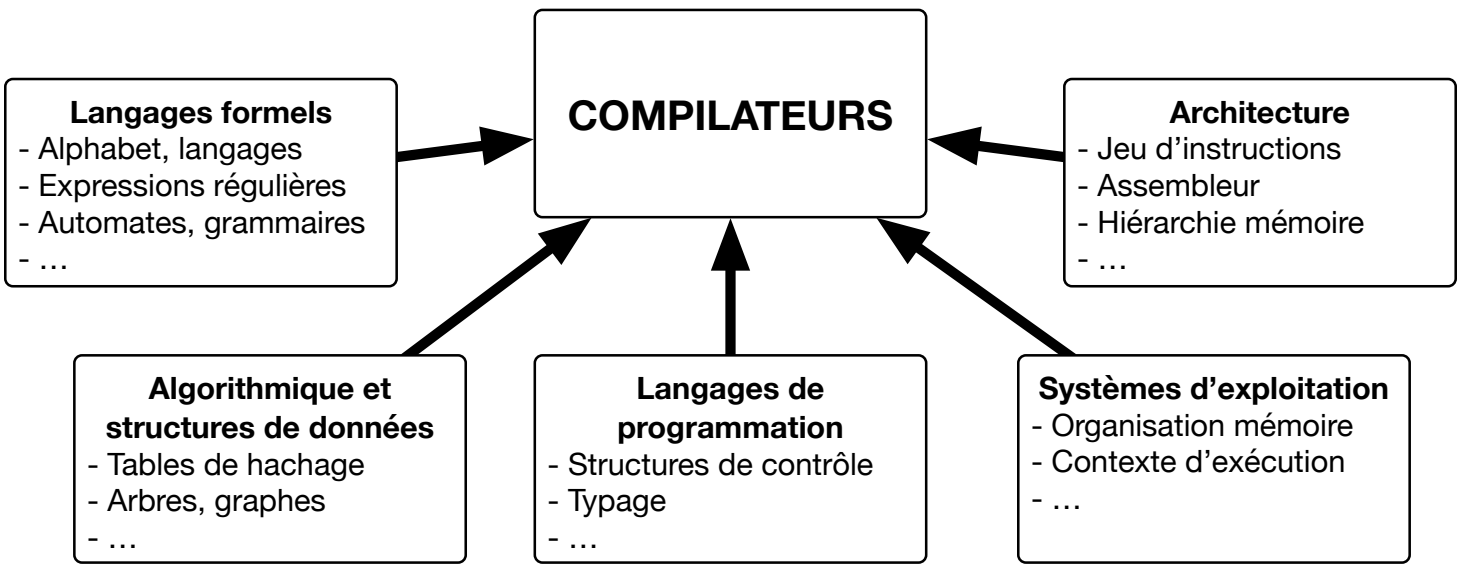


Compilation vs interprétation

- Le compilateur peut produire un code directement exécutable fonctionnant pour toutes les données en entrée
  - Processus de traduction complexe
  - Code produit généralement très efficace
  - Optimisation principalement durant la compilation (*statique*)
- L'interpréteur est le code directement exécutable qui recalcule le résultat à chaque fois
  - Processus de traduction simple
  - Calcul du résultat généralement peu efficace
  - Optimisation exclusivement durant l'exécution (*dynamique*)

Pourquoi étudier la compilation ?

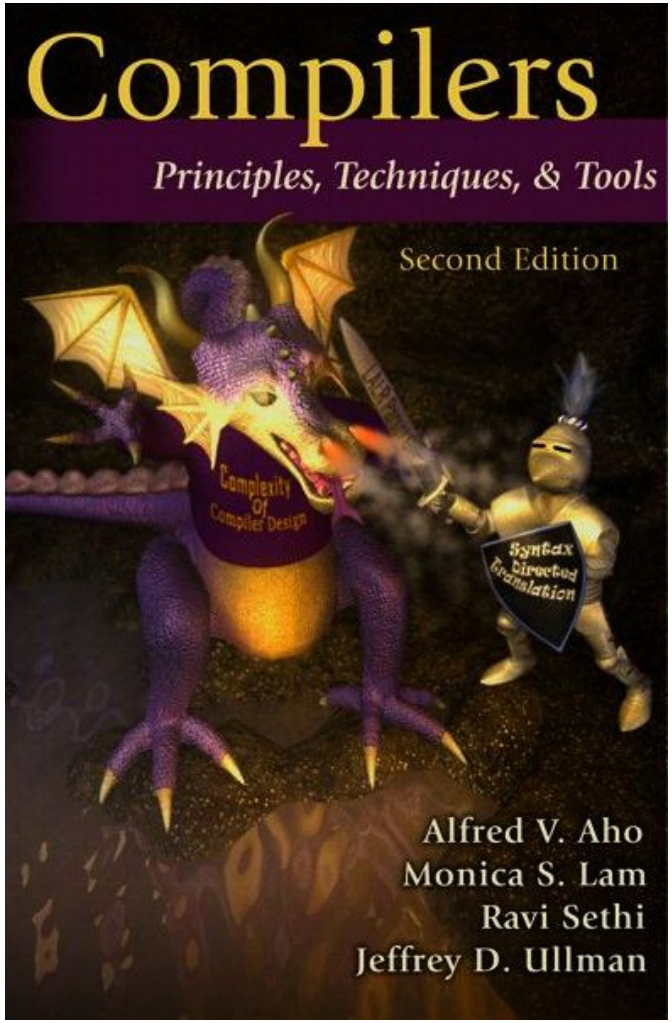
- Écrire un compilateur : rite de passage des informaticiens
  - Sollicite des prérequis de la théorie à la technique avancée
  - Apporte une compréhension générale de la programmation
  - Forme à des techniques et outils largement réutilisables
  - Constitue un projet conséquent et relativement difficile

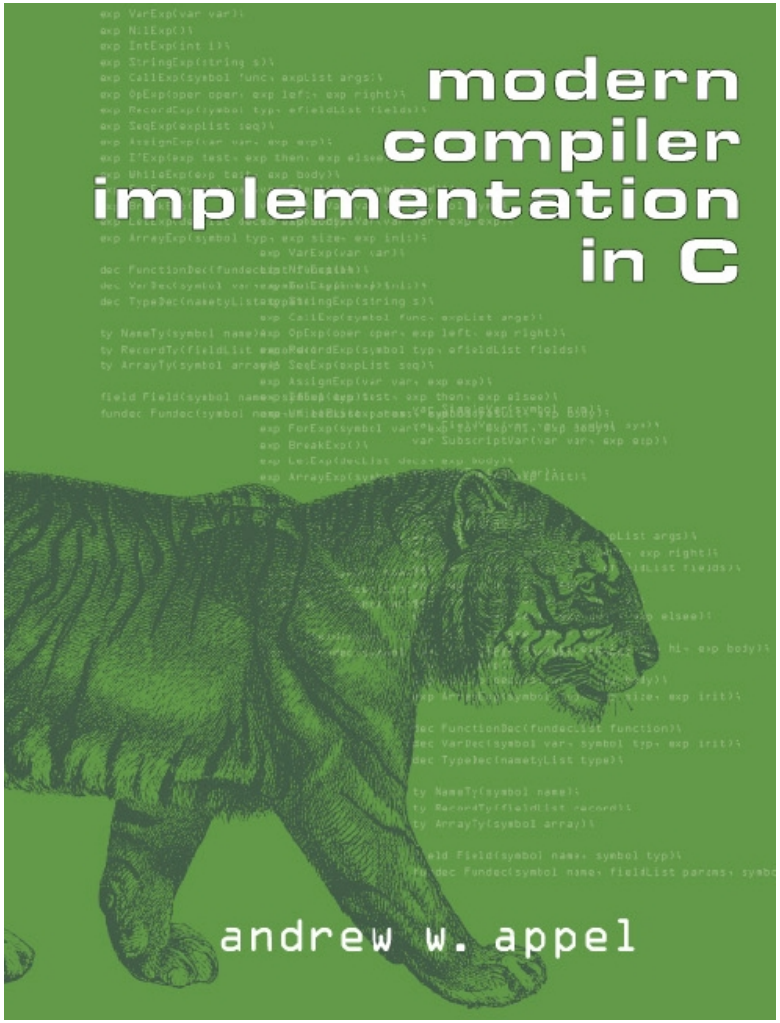


Exemples de problèmes pratiques

- Passer de fichiers d'entrée à des structures de données
- Passer de structures de données à des fichiers de sortie
- Formater un code selon un style défini
- Générer la documentation technique d'un code existant
- Appliquer une tâche répétitive à une base de code
- Faire de la coloration syntaxique pour votre format préféré
- Repérer des constructions particulières dans un code
- Créer le langage/format spécialisé idéal pour votre activité
- Réutiliser l'infrastructure d'un compilateur existant
  - Pour compiler depuis votre langage particulier
  - Pour compiler vers votre architecture particulière

Références


Compilers
Principles, Techniques, & Tools
Second Edition
Complexity of Compiler Design
Syntax directed translation
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman


modern compiler implementation in C
andrew w. appel

Qualités d'un compilateur

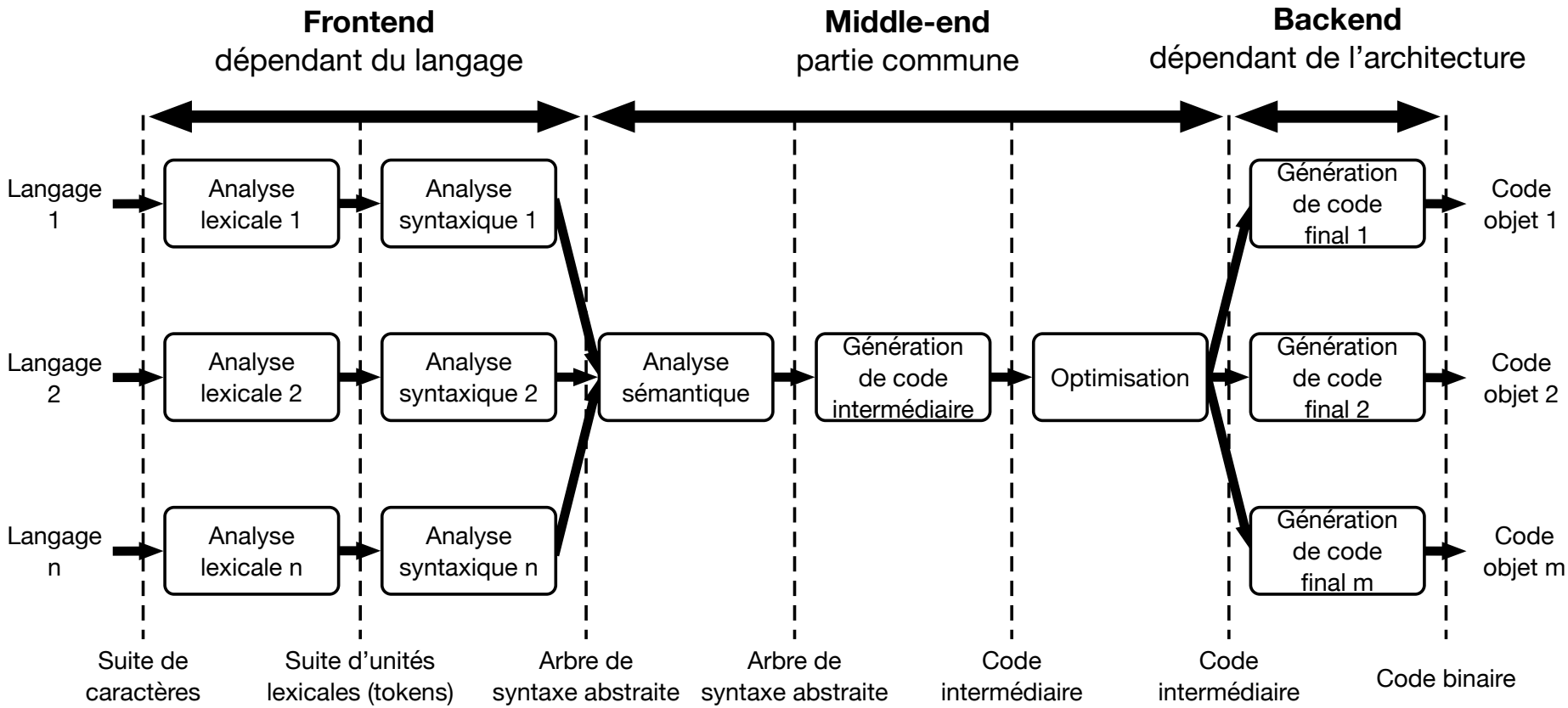
Plus important

- Correction : le programme compilé doit avant tout correspondre au même calcul que le programme original
- Optimisation : le programme généré doit être rapide
- Efficacité : le compilateur lui-même doit être rapide
- Productivité : le compilateur doit signaler les erreurs et les risques de manière compréhensible
- Portabilité : le compilateur doit être extensible facilement à de nouveaux langages source et architectures cibles
- Prévisibilité : les optimisations doivent être cohérentes et prévisibles (hahaha)

Moins important

Architecture des compilateurs modernes

- Traitement organisé en phases
- Grande modularité pour maintenir, réutiliser et étendre



Phases de la compilation

1. **Analyse lexicale** : traduit un flot de caractères en un flot d'unités lexicales représentant des suites de caractères
2. **Analyse syntaxique** : vérifie que la suite correspond à une construction permise et produit l'arbre de syntaxe abstraite
3. **Analyse sémantique** : effectue les vérifications sur la sémantique du programme (typage, résolution des noms...)
4. **Génération de code intermédiaire** : traduit l'arbre de syntaxe abstraite en un code pour une machine abstraite
5. **Optimisation** : tente d'améliorer le code intermédiaire
6. **Génération de code final** : traduit le code intermédiaire générique en code natif dépendant de l'architecture cible

Principaux compilateurs libres

- GCC, GNU Compiler Collection
  - Créé en 1987 par le projet GNU
  - Langages : C/C++/Java/Objective-C/Fortran etc.
  - Architectures cibles : plus de 20
  - Représentations : GIMPLE, GENERIC, RTL
  - Plus de 15 millions de lignes de code (Linux : 19)

- Clang/LLVM
  - Créé en 2005, rendu libre en 2007 par Apple
  - Langages : C/C++/Objective-C (CUDA/OpenCL)
  - Architectures cibles : x86 et ARM
  - Représentations : LLVM-IR, Bitcode
  - Plus de 4 millions de lignes de code

Exemple : compilation du langage Arith

- Langage source : « Arith » expressions arithmétiques

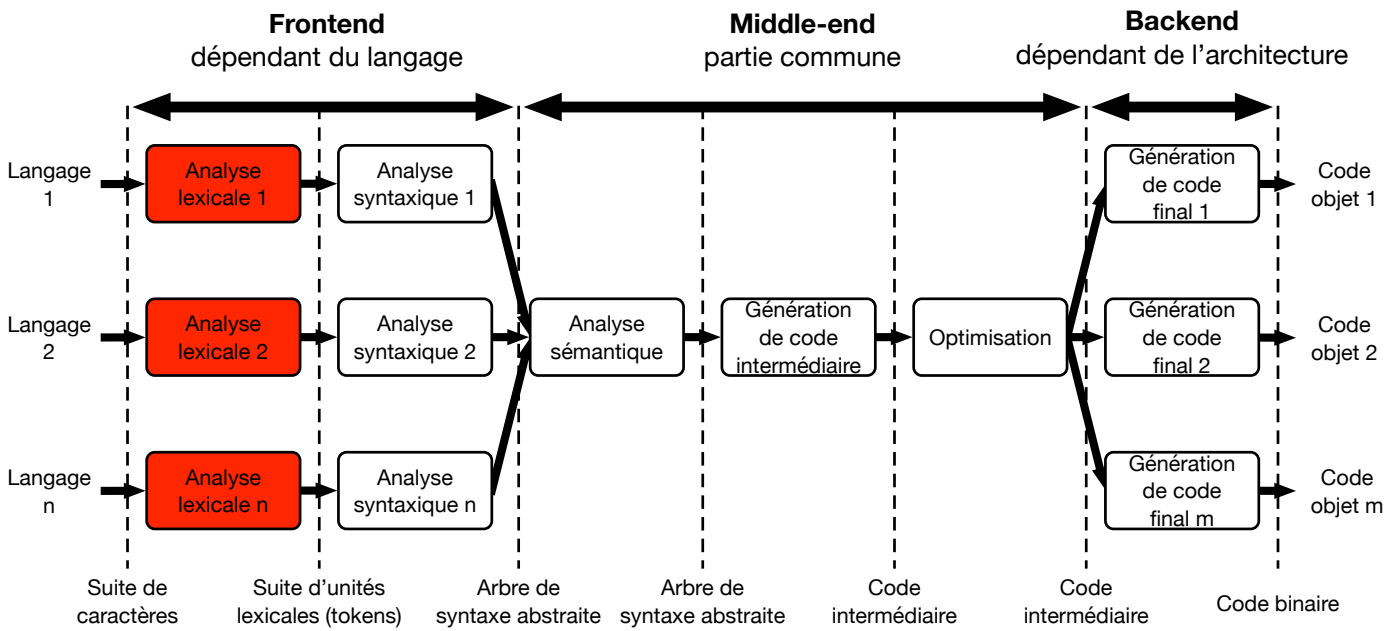
Exemple de code source Arith

```
x = 12;
y = (5 + x) * 10 + 3;
print x + y;
```

- Langage cible : langage machine d'une architecture à pile

Instruction	Description
`PUSH M`	Empile la donnée à l'adresse `M`
`POP M`	Dépile et place la donnée dépilée à l'adresse `M`
`ADD`	Dépile deux données, fait leur addition et empile le résultat
`MUL`	Dépile deux données, fait leur multiplication et empile le résultat
`PRINT`	Affiche la donnée en sommet de pile

Analyse lexicale



Notions de langages formels

Vocabulaire

- Alphabet : ensemble fini de symboles
- Mot : séquence finie de symboles d'un alphabet (vide : ε)
- Langage : ensemble de mots

Expressions rationnelles

- Décrit un ensemble de mots possibles selon une syntaxe

Un caractère	
c	le caractère c
$[c_1 c_2 ... c_n]$	c_1 ou c_2 ... ou c_n
$[c_1 - c_2]$	un caractère entre c_1 et c_2 compris
Une séquence de caractères	
$\alpha\beta$	concaténation
$\alpha\|\beta$	alternative
$\alpha*$	répétition zero ou plusieurs fois
$\alpha+$	répétition au moins une fois

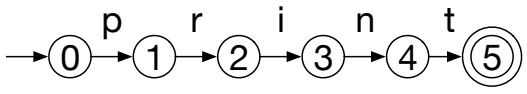
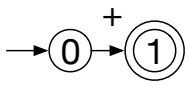
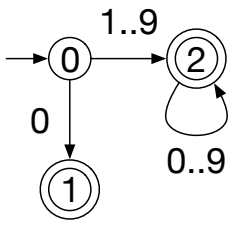
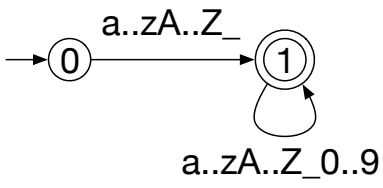
Analyse lexicale

L'analyse lexicale découpe le texte du code source en « mots » appelés « tokens » pour faciliter le travail de la phase suivante

- Décrit les tokens avec des expressions rationnelles
- Utilise des automates finis pour les reconnaître
  - Générés à partir des expressions rationnelles
- Ignore le texte superflu (commentaires, espaces)
- Émet un couple *(type, valeur)* et réalise éventuellement des actions supplémentaires pour chaque token
  - Exemples : (identificateur, i) ; (nombre, 42)
  - Les types des tokens sont les symboles terminaux de la grammaire du langage

L'analyseur lexical est un automate fini pour l'union de toutes les expressions régulières définissant les tokens

Reconnaissance des tokens

Token	Expression rationnelle	Automate
mot clé `print`	`print`	
symbole « + »	`+`	
nombre	`0|[1-9][0-9]*`	
identificateur	`[a-zA-Z_][a-zA-Z_0-9]*`	

Générateur d'analyseur lexical Lex

- Créé en 1975 par AT&T pour Unix, version libre Flex (1987)
- Lit une spécification de l'analyseur et génére son code C
- Utilise une spécification en 3 parties séparées par « %% »
  1. **Déclarations** :
     - Déclarations C entre « %{ » et « %} »
     - Options et commandes spécifiques à Lex
     - Définitions d'expressions rationnelles
  2. **Règles de traduction** : suite ordonnée de couples « `regex` {`action`} » où `regex` est une expression rationnelle Lex et `action` est un bloc de code C
  3. **Fonctions auxilliaires** : définitions de fonctions C

Structure d'un ficher Lex (extension .l)

```
/* Section des déclarations C et/ou Lex */
%%
/* Section des règles de traduction */
%%
/* Section des fonctions C */
```

Fonctionnement de Lex

Génère une fonction `yylex()` lançant l'analyse lexicale sur le fichier pointé par la variable globale `yyin` de type `FILE*`

- Reconnaît le plus long mot correspondant à une regex
- Choisit la première regex dans la liste en cas d'égalité
- Exécute l'action associée à la regex, où on a accès à :
  - `yytext` : la chaine correspondant au mot reconnu
  - `yyleng` : la taille du mot reconnu
  - `yylval` : la valeur à associer au token
- Affiche sur la sortie standard les caractères non reconnus
- Termine dans un des cas suivants (hors erreur) :
  - le fichier est parcouru entièrement
  - une action effectue `return` (retourne le code du token)
  - la fonction `yyterminate()` est appelée

Exemple : compilation du langage Arith

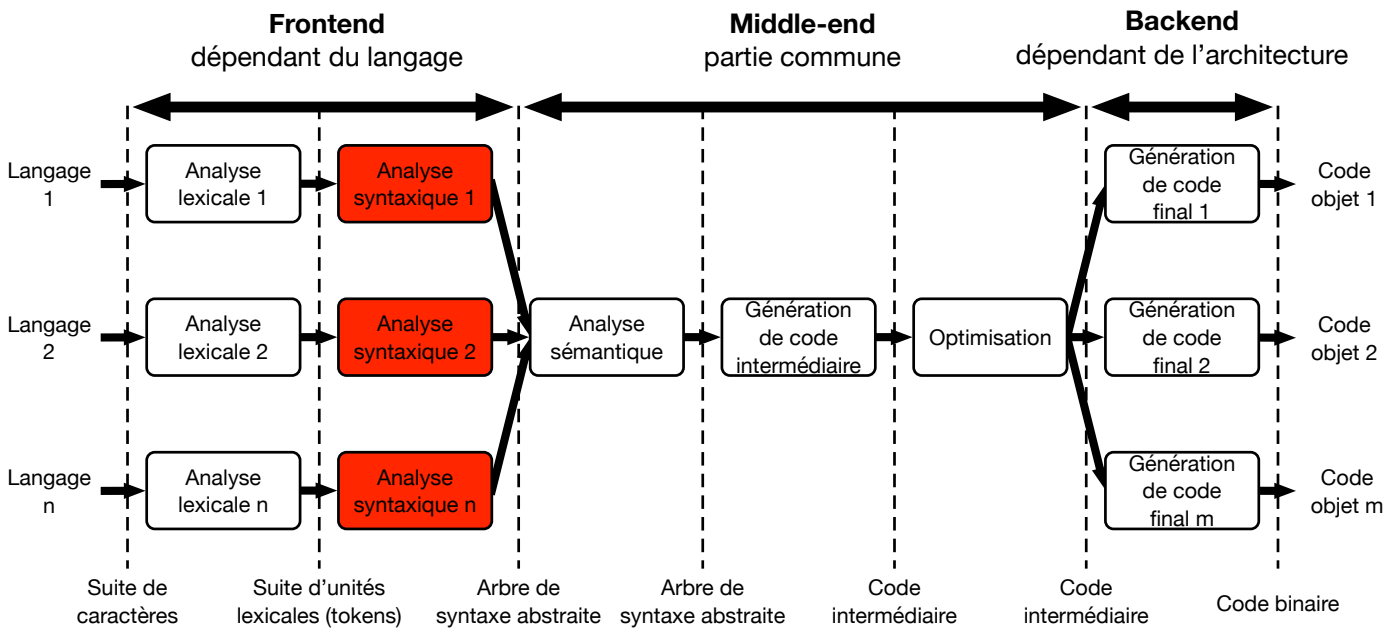
Code Lex pour l'analyse lexicale du langage Arith [code]

```
%{
  #include <stdio.h>
  #include <stdlib.h>
%}

identifier [a-zA-Z_]+[0-9a-zA-Z_]*
number     [0-9]+

%%

print         { printf("[L] keyword:    %s\n", yytext); }
{identifier}  { printf("[L] identifier: %s\n", yytext); }
{number}      { printf("[L] number:     %s\n", yytext); }
[()+*=;]      { printf("[L] symbol:     '%s'\n", yytext); }
[ \t\n]       { }
.             { printf("[L] ERROR: unknown character %s\n", yytext); }

%%

int main() {
  yylex();
  return 0;
}
```

L'analyse lexicale aujourd'hui

- Dans la vie d'informaticien de tous les jours :
  - Extrêmement utile
  - Lex existe en Java (JLex), OCaml (ocamllex), Python (PLY)
  - Couteau suisse pour le traitement des textes
- Dans la recherche :
  - 1960 : algorithme de conversion des expressions rationnelles en automates finis déterministes par McNaughton et Yamada
  - Théorie suffisamment solide et performante
  - Recherche très active en amont au niveau des langages

Analyse syntaxique



Notions de langages formels

- Grammaire : spécification de la structure syntaxique d'un langage, composée d'un ensemble de *productions* mettant en jeu des *terminaux* et *non-terminaux*
- Terminal : symbole élémentaire du langage (token)
- Non-terminal : variable représentant une liste de terminaux
- Axiome : non-terminal représentant le symbole de départ
- Production : règle de réécriture pour un non-terminal
  - Notée « *partie gauche* $\rightarrow$ *partie droite* » où
  - *partie gauche* est un non-terminal
  - *partie droite* est une liste de terminaux et non-terminaux
- Dérivation : application d'une production
  - Notée « *partie gauche* $\Rightarrow$ *partie droite* » où
  - *partie gauche* est une liste de terminaux et non-terminaux
  - *partie droite* est la liste après application de la réécriture

Grammaire d'expressions arithmétiques

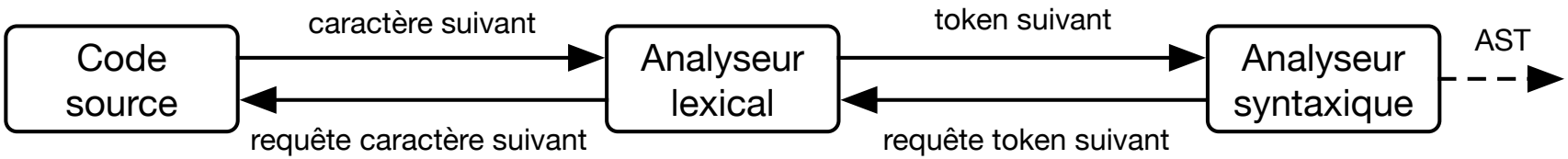
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & F \\
F & \rightarrow & \texttt{(}\ E\ \texttt{)} \\
F & \rightarrow & \texttt{identificateur} \\
F & \rightarrow & \texttt{nombre}
\end{array}
$$

- Liste de 7 productions
- Axiome : E, apparait en premier par convention
- Non-terminaux : E, T, F
- Terminaux : `+`, `*`, `(`, `)`, `identificateur`, `nombre`
- Exemples de dérivations :
  $E \Rightarrow T \Rightarrow F \Rightarrow$ `(` E `)` $\Rightarrow$ `(` T `)` $\Rightarrow$ `(` F `)` $\Rightarrow$ `(` `nombre` `)`

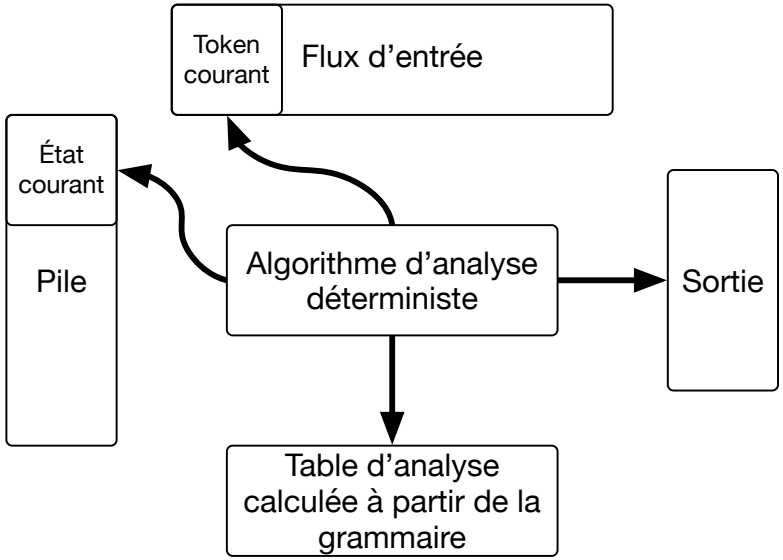
Analyse syntaxique

L'analyse syntaxique cherche dans une grammaire une séquence de dérivations de l'axiome jusqu'à la liste de tokens en entrée

- Décrit la syntaxe avec une grammaire
- Utilise un automate à pile pour trouver les dérivations
  - Généré à partir de la grammaire
- Exécute éventuellement des actions à chaque dérivation
  - Chaque terminal et non-terminal peut porter une valeur
  - Peut construire l'arbre de syntaxe abstrait durant l'analyse
- Émet une erreur de syntaxe si la séquence n'existe pas
- Demande les tokens au fur et à mesure à l'analyseur lexical



Reconnaissance des listes de dérivations



- Automate à pile analysant le flux de tokens en entrée
  - L'algorithme et la table dépendent du type de grammaire
  - La table se construit directement à partir de la grammaire
- La sortie dépend des actions associées aux dérivations
  - Arbre de dérivation : représente une liste de dérivations
  - Arbre de syntaxe abstraite : représente le code utile
  - Résultat direct (calcul, code intermédiaire...)

Générateur d'analyseur syntaxique Yacc

- Créé en 1970 par AT&T pour Unix, version libre Bison (1988)
- Lit une spécification de l'analyseur et génére son code C
- Utilise une spécification en 3 parties séparées par « %% »
  1. **Déclarations** :
     - Déclarations C entre « %{ » et « %} »
     - Déclaration des tokens et du type de leurs valeurs
     - Options et commandes spécifiques à Yacc
  2. **Règles de traduction** : suite ordonnée de couples « `production` {`action`} » où `production` est une production Yacc et `action` est un bloc de code C
  3. **Fonctions auxilliaires** : définitions de fonctions C

Structure d'un ficher Yacc (extension .y)

```
/* Section des déclarations C et/ou Lex */
%%
/* Section des règles de traduction */
%%
/* Section des fonctions C */
```

Fonctionnement de Yacc

Génère une fonction `yyparse()` lançant l'analyse syntaxique sur le fichier pointé par la variable globale `yyin` de type `FILE*`

- Reconnaît les productions selon un mode « LALR »
  - *ascendant* : des feuilles de l'arbre de dérivation à l'axiome
  - *dérivation la plus à droite* : cherche les dérivations de l'élément le plus à droite des productions d'abord
- Exécute l'action associée à la production, où on a :
  - `$$` : la valeur associée au non-terminal de gauche
  - `$`i : la valeur associée au $i^{ème}$ élément de la partie droite
- Termine dans un des cas suivants (hors erreur) :
  - le fichier est parcouru entièrement
  - une action effectue `return`
  - la macro `YYABORT` ou `YYERROR` ou `YYACCEPT` est utilisée

Exemple : compilation du langage Arith

- Le code Lex retourne les tokens
- Code conforme si Yacc peut remonter à l'axiome

Extrait de code Yacc pour l'analyse syntaxique d'Arith [code]

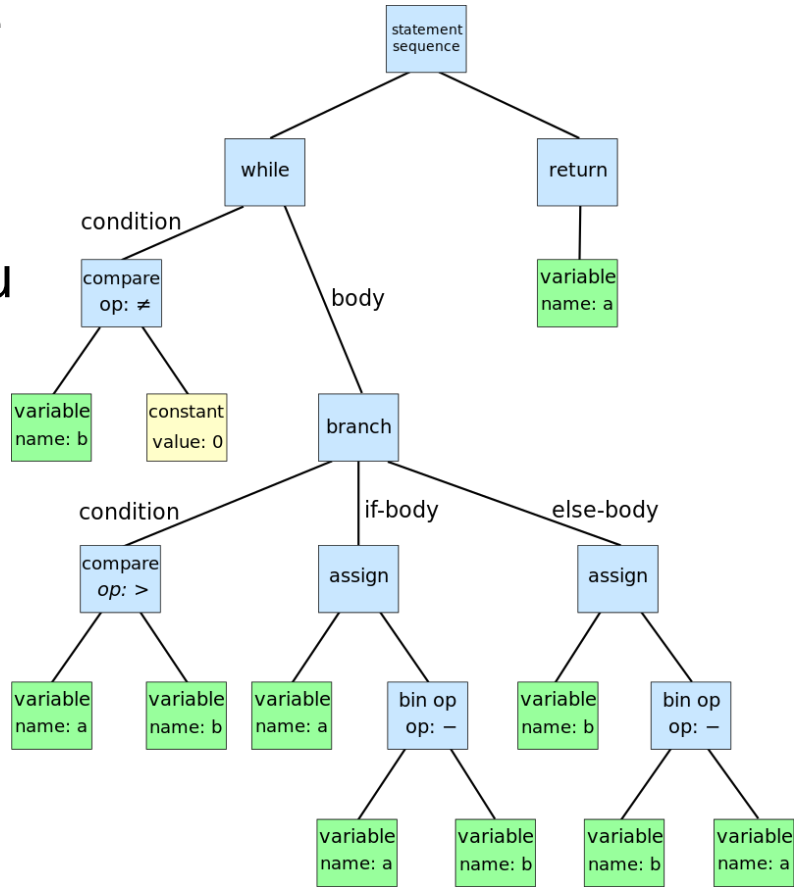
```
%%
axiom:
    statement_list                  { printf("[Y] Match :-) !\n"); return 0; }
  ;
statement_list:
    statement statement_list        { printf("[Y] stmt_lst -> stmt stmt_lst\n"); }
  | statement                       { printf("[Y] stmt_lst -> stmt\n"); }
  ;
statement:
    IDENTIFIER '=' expression ';'{ printf("[Y] stmt -> ID (%s) = expr;\n",$1);}
  | PRINT expression ';'            { printf("[Y] stmt -> PRINT expr;\n"); }
  ;
expression:
    expression '+' expression       { printf("[Y] expr -> expr + expr\n"); }
  | expression '*' expression       { printf("[Y] expr -> expr * expr\n"); }
  | '(' expression ')'              { printf("[Y] expr -> ( expr )\n"); }
  | IDENTIFIER                      { printf("[Y] expr -> ID (%s)\n", $1); }
  | NUMBER                          { printf("[Y] expr -> NUMBER (%d)\n", $1); }
  ;
%%
```

Arbre de syntaxe abstraite (AST)

Représentation de la structure syntaxique d'un code sous la forme d'une structure de données d'arbre

- AST construit durant l'analyse syntaxique par les actions
- Chaque action peut créer un nouveau noeud (opérateur) ou feuille (opérande)
- Yacc : construction depuis les feuilles vers la racine

```
while (b != 0) {
  if (a > b)
    a = a - b;
  else
    b = b - a;
}
return a;
```



Exemple : compilation du langage Arith

- Les attributs des non-terminaux sont des noeuds de l'AST
- L'AST est construit durant l'analyse de Yacc

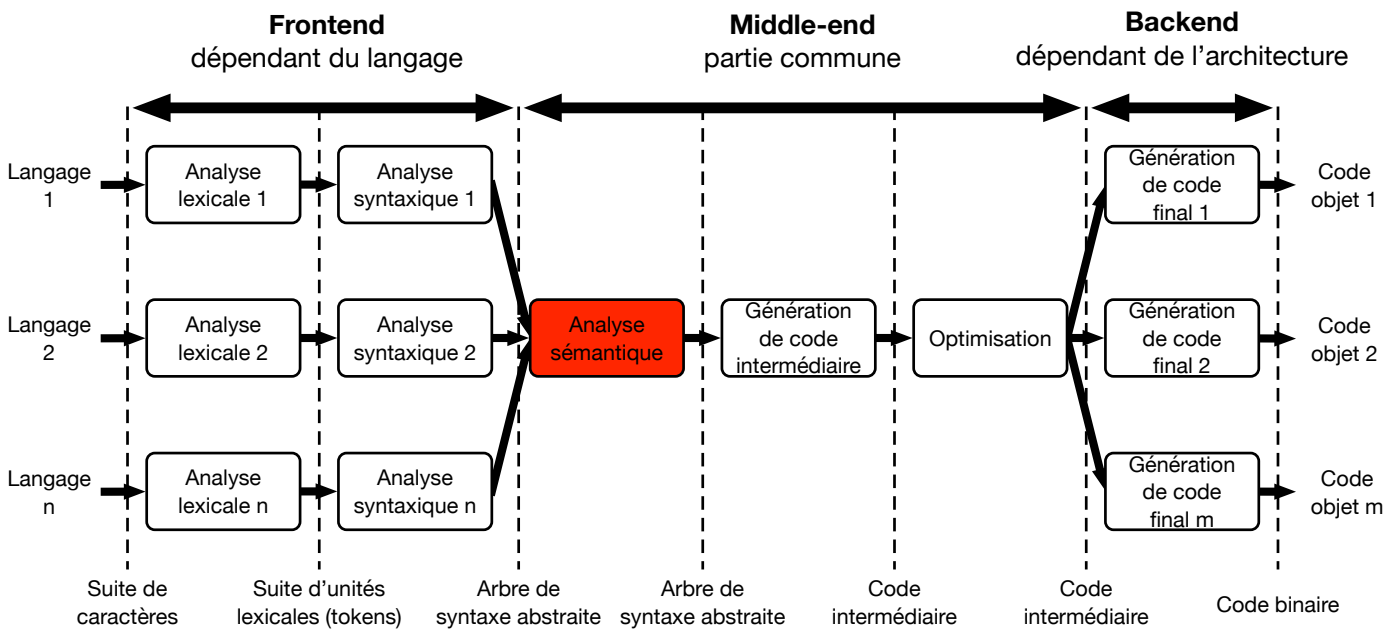
Extrait de code Yacc pour l'analyse syntaxique d'Arith [code]

```
%%
axiom:
    statement_list                  { parser_ast = $1; return 0; }
  ;
statement_list:
    statement statement_list        { $$ = ast_concat($1, $2); }
  | statement                       { $$ = $1; }
  ;
statement:
    IDENTIFIER '=' expression ';'{ $$ = ast_new_statement($1, $3); }
  | PRINT expression ';'            { $$ = ast_new_statement(NULL, $2); }
  ;
expression:
    expression '+' expression       { $$ = ast_new_operation(ast_type_add,$1,$3);}
  | expression '*' expression       { $$ = ast_new_operation(ast_type_mul,$1,$3);}
  | '(' expression ')'              { $$ = $2; }
  | IDENTIFIER                      { $$ = ast_new_identifier($1); }
  | NUMBER                          { $$ = ast_new_number($1); }
  ;
%%
```

L'analyse syntaxique aujourd'hui

- Dans la vie d'informaticien de tous les jours :
  - Yacc en Java (CUP), OCaml (ocamlyacc), Python (PLY)
  - Passage de données formatées en structures de données
  - Couteau suisse universel pour le traitement des textes
- Dans la recherche :
  - 1962-1963 démonstration d'équivalence entre grammaire à contexte libre et grammaires reconnues par les automates à piles par Chomsky, Evey, et Schutzenberger (indépendamment)
  - Théorie solide et performante depuis ces années
  - Recherche active au niveau de l'arbre de syntaxe abstraite
    - Représentation des langages parallèles
    - Traitement rapide, extraction d'informations

Analyse sémantique



Analyse sémantique

L'analyse sémantique effectue les vérifications nécessaires pour s'assurer que le programme appartient bien au langage

- Construit la table des symboles
  - Informations (type, portée etc.) relatives aux identificateurs
- Assure qu'un nom n'est pas associé à plus d'une variable
- Vérifie la cohérence des types dans les expressions
- Contrôle l'initialisation des variables avant leur utilisation
- Ajoute des informations à l'arbre de syntaxe abstraite

Généralement inutile au quotidien de l'informaticien mais domaine très actif de la recherche :

- Solutions à l'équivalence de types
- Systèmes de typages forts, cohérents et efficaces
- Compilation certifiée

Exemple : compilation du langage Arith

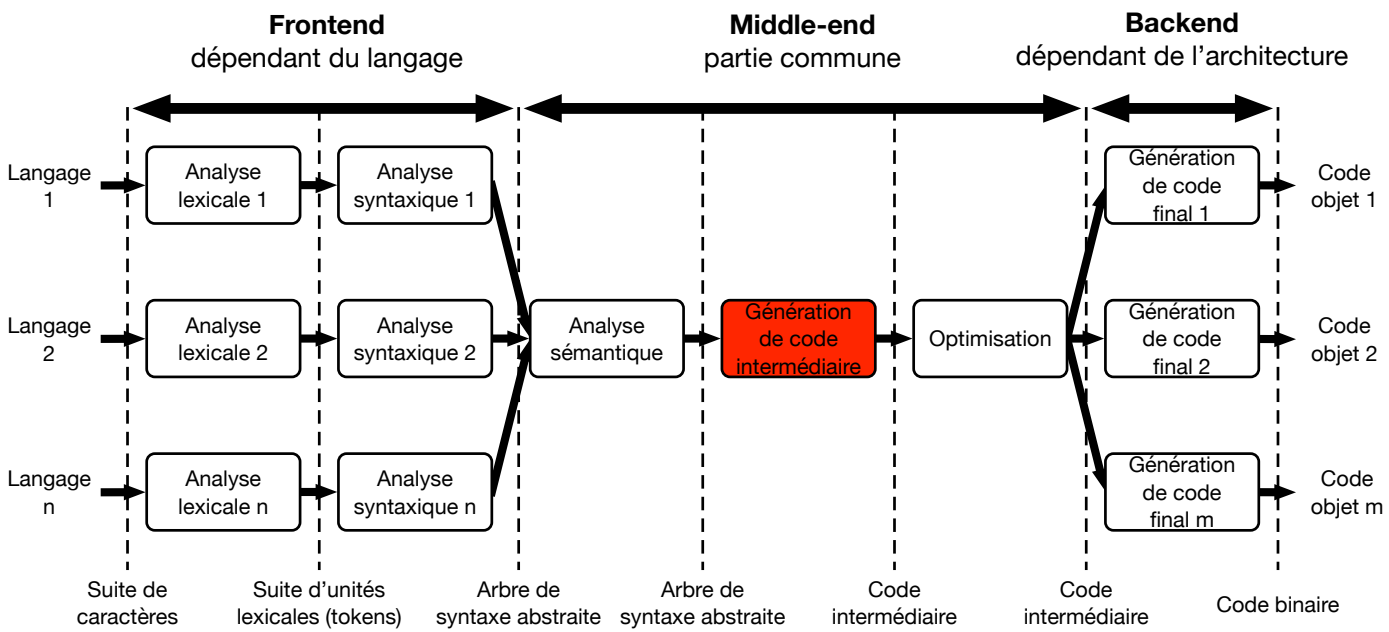
Construction de la table des symboles

- Parcours récursif de l'arbre de syntaxe abstraite
- Détection de l'utilisation de variables non initialisées

Extrait de code de l'analyse sémantique d'Arith [code]

```
void symbol_build_table(struct ast* ast, struct symbol** symbol_table) {
  struct symbol* identifier;
  do {
    switch (ast->type) {
      case ast_type_identifier:
        identifier = symbol_lookup(*symbol_table, ast->u.identifier);
        if (identifier == NULL) {
          fprintf(stderr, "[S] SEMANTIC ERROR: %s is not initialized\n",
                  ast->u.identifier);
          exit(1);
        }
        ast->type = ast_type_symbol;
        ast->u.symbol = identifier;
        break;
      case ast_type_add:
        symbol_build_table(ast->u.operation.left,  symbol_table);
        symbol_build_table(ast->u.operation.right, symbol_table);
        break;

        ...
```

Génération de code intermédiaire



Génération de code intermédiaire

La génération de code intermédiaire traduit l'arbre de syntaxe abstraite en un code pour une machine abstraite

- Abaisse le niveau d'abstraction en restant portable
- Met le code sous une forme adaptée à certaines analyses
- Facilite les optimisations indépendantes de l'architecture
- Opère un choix critique entre plusieurs formes :
  - Représentation structurée donnant une vision globale (exemple : arbres ou graphes)
  - Représentation linéaire simple à manipuler (exemple : code trois adresses)
  - Hybride (exemple : graphe de flot de contrôle)

Code trois adresses

Representation intermédiaire linéaire formée de code à trois adresses, c'est à dire d'instructions de la forme :

$$x \leftarrow y \text{ op } z$$

avec un seul opérateur (op) et au plus trois opérandes (x, y et z)

- Forme intermédiaire abstraite sans limite mémoire
- Structure de code simple (liste d'instructions)
- Instructions simples, élémentaires et générales
- Les opérandes renvoient à une table des symboles
  - Variables du programme (locales, paramètres...)
  - Variables temporaire pour les résultats intermédiaires
  - Labels définissant la cible de branchements
  - Constantes

Instructions d'un code trois adresses

Affectation	$x \leftarrow y$ op z	avec op $\in \{+, -, *, /, ...\}$
	$x \leftarrow$ op y	avec op $\in \{-, !, sin, cos, ...\}$
	$x \leftarrow y$	
Branchement	goto L	avec L un label
	if x relop y goto L	avec relop $\in \{>, <, \geq, \leq, ...\}$
Appel de procédure	param x_1 param x_2 ... param x_n call p n	appel de procédure p avec n paramètres
Adressage indicé	$x[i] \leftarrow y$	
	$x \leftarrow y[i]$	
Adressage direct	$x \leftarrow \&y$	
	$x \leftarrow *y$	
	$*x \leftarrow y$	

Exemple : compilation du langage Arith

- Structure de données de « quad »

Structure de données de quad

```
struct quad {
  char op;              // Operator
  struct symbol* arg1; // First argument
  struct symbol* arg2; // Second argument
  struct symbol* res;  // Result
  struct quad*   next; // Next quad in the NULL-terminated linked list
};
```

- Génération des quads par parcours récursif de l'AST

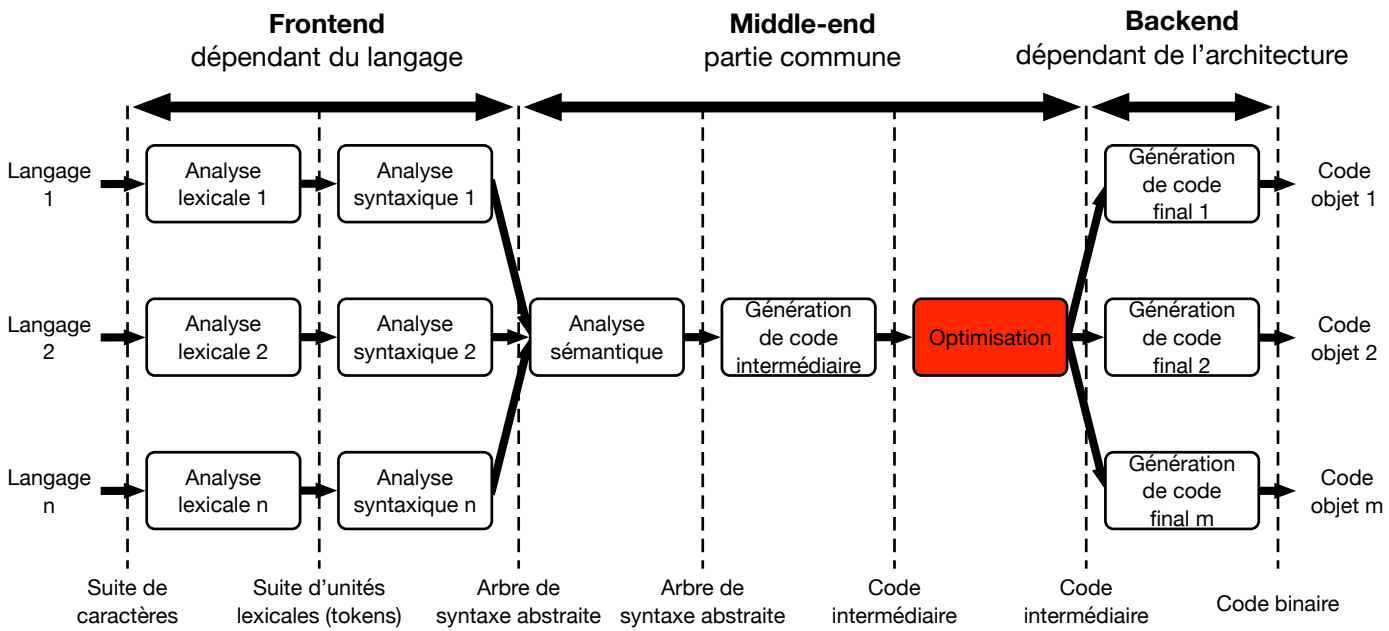
Extrait de la génération de code pour Arith [code]

```
struct codegen* codegen_ast(struct ast* ast, struct symbol** symbol_table) {
  ...
    switch (ast->type) {
      case ast_type_add:
        left  = codegen_ast(ast->u.operation.left,  symbol_table);
        right = codegen_ast(ast->u.operation.right, symbol_table);
        new_code = quad_gen('+', left->result, right->result, cg->result);
        cg->result = symbol_new_temp(symbol_table);
        cg->code = left->code;
        quad_add(&cg->code, right->code);
        quad_add(&cg->code, new_code);
        ...
```

La génération de CI aujourd'hui

- Dans la vie d'informaticien de tous les jours :
  - Inutile en dehors de la compilation
- Dans la recherche :
  - 1970 : graphe de flot de contrôle (CFG) par Frances Allen
  - 198x : single assignment form (SSA) par Ron Cytron et al.
  - 1998 : array SSA par Kathleen Knobe et Vivek Sarkar
  - Représentations plus riches, support du parallélisme
  - Formes adaptées à la compilation *just in time*

Optimisation



Optimisation de code intermédiaire

L'optimisation de code vise à améliorer sa qualité

- Trouve une « meilleure » forme de même sémantique
  - Réduction du temps d'exécution
  - Réduction de l'empreinte mémoire
  - Réduction de l'énergie consommée, etc.
- Cible des aspects indépendants de l'architecture
  - Réduction du nombre d'instructions, d'accès mémoire
  - Parallélisation, etc.
- Utilise des techniques d'optimisation
  - Une technique focalise sur un nombre restreint d'aspects
  - Une technique peut aussi détériorer la performance
  - Une technique peut être utilisée plusieurs fois
  - Choisir ou ordonner les techniques est indécidable

Exemples de techniques d'optimisation

- **Réduction de force**
  - Consiste à utiliser des opérations moins coûteuses

```
y = x * 2;
```
→
```
y = x << 1;
```

- **Simplifications algébriques**

```
y = x * 1;
y = 0 + x;
```
→
```
y = x;
y = x;
```

- **Pliage de constantes**
  - Consiste à calculer la valeur des expressions constantes

```
x = 4 + 2;
```
→
```
x = 6;
```

  - Attention à la sémantique du langage : `x` short en C

```
x = 32767 + 1;
```
→
```
x = -32768;
```

Exemples de techniques d'optimisation

- **Propagation de constantes / de copies**
  - Consiste à éviter les affectations inutiles

```
x = 42;          →   x = 42;
y = x + 17;          y = 42 + 17;
```

  - Si `x` n'est pas utilisée après, on peut supprimer `x = 42;`
  - Peut permettre le pliage de constantes (et inversement)
  - Même principe pour éviter les copies :

```
y = x;           →   y = x;
z = y + t            z = x + t
```

- **Élimination des sous-expressions communes**
  - Consiste à remplacer les calculs par des copies

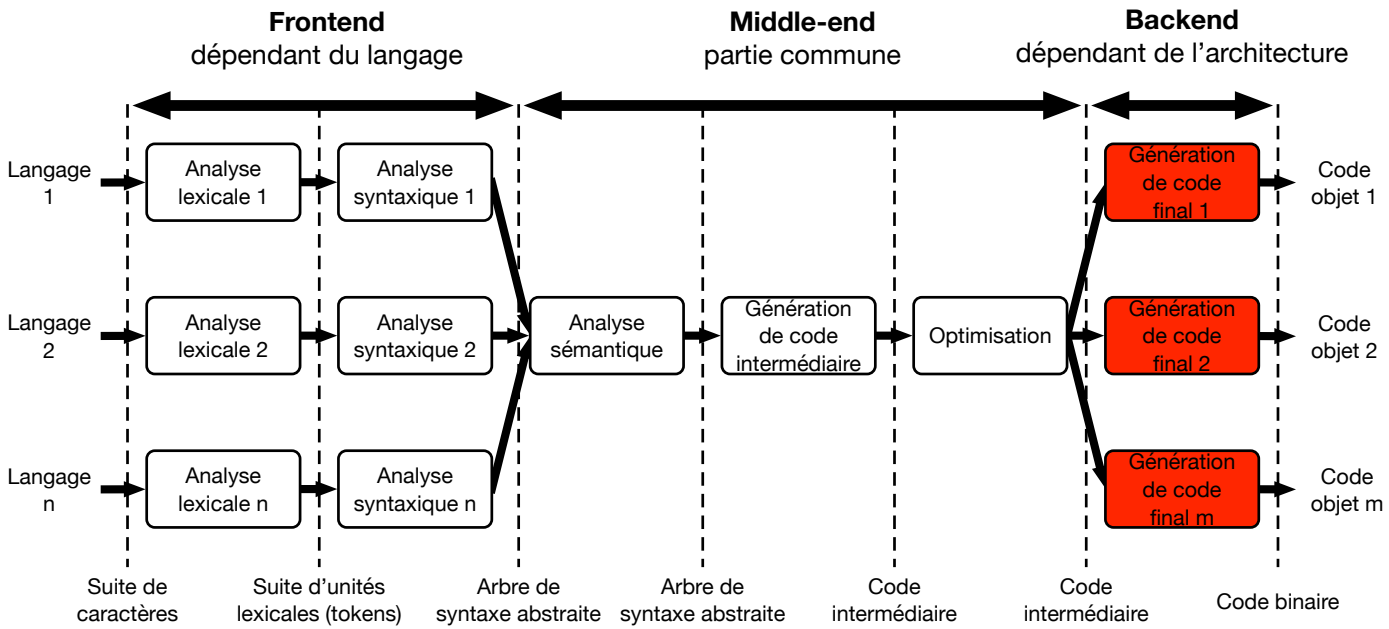
```
y = a + b;       →   y = a + b;
...                  ...
z = a + b;           z = y;
```

  - `a`, `b` et `y` ne doivent pas avoir été modifiées

L'optimisation de code aujourd'hui

- Dans la vie d'informaticien de tous les jours :
  - Techniques utiles pour créer des applications performantes
  - Mais toujours laisser au compilateur ce qu'il sait bien faire
  - *Premature optimization is the root of all evil*
- Dans la recherche :
  - La branche la plus active de la compilation
  - Analyses précises du code
  - Sélection et ordonnancement des phases d'optimisation
  - Parallélisation automatique
  - Utilisation optimisée de la hiérarchie mémoire
  - Optimisation à la volée
  - Reconnaissance d'algorithmes
  - Présentation à venir !

Génération de code final



Génération de code final

La génération de code final traduit le langage intermédiaire en un langage machine dépendant de l'architecture

- Traduit les instructions abstraites en instructions machine
  - Dépend du jeu d'instruction de l'architecture cible
- Réalise les optimisations dépendantes de l'architecture
  - Gestion des registres
  - Réduction de force spécifique
  - Utilisation des mémoires cache
  - Pipeline, parallélisme d'instruction, vectorisation etc.

- Il faudra me réinviter pour un thème architecture !

Exemple : compilation du langage Arith

- Traduction systématique des instructions finales
- La table des symboles donne l'état initial de la mémoire

Extrait de la génération de code final pour Arith [code]

```
struct instruction* instruction_gen_add(struct quad* quad) {
  struct instruction* list = NULL;
  instruction_add(&list, instruction_gen(instruction_type_push, quad->arg1));
  instruction_add(&list, instruction_gen(instruction_type_push, quad->arg2));
  instruction_add(&list, instruction_gen(instruction_type_add, NULL));
  instruction_add(&list, instruction_gen(instruction_type_pop, quad->res));
  return list;
}
```

Conclusion

- Les compilateurs sont des outils complexes
  - Grand nombre de technologies mises en jeu
  - Modularité extrême pour des raisons de coût
  - Savant mélange de rigueur formelle et d'alchimie
- Comprendre les compilateurs est un atout
  - Compréhension des langages et des formats
  - Connaissance d'outils de manipulation de textes
  - Optimisation des programmes
- Beaucoup de problèmes ouverts au niveau recherche
  - Support des avancées des langages et des architectures
  - Sémantique, compilation certifiée
  - Optimisation, optimisation, optimisation