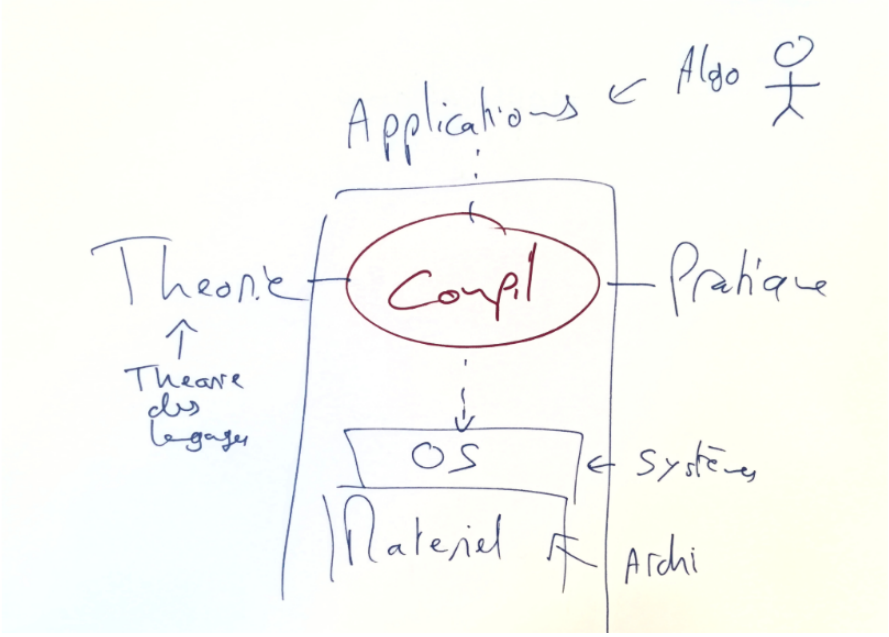
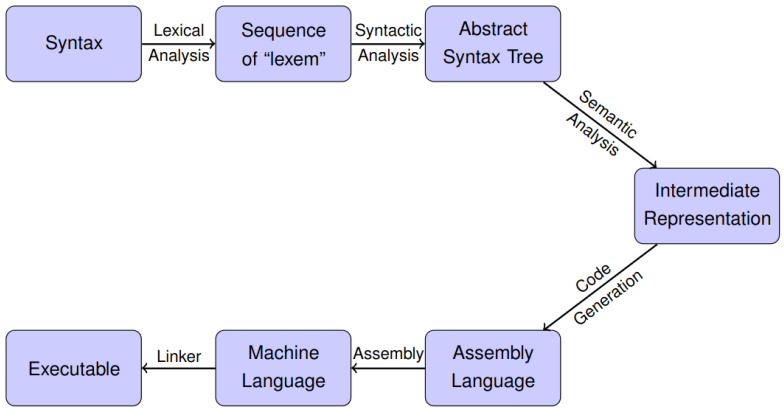
1 Introduction

Où on est ?



C'est quoi un compilateur ? Dessin : bout de code → binaire (et erreurs).

Comment ça marche ?



Analyse lexicale But: transformer le code en une suite de unités élémentaires (lexèmes).

Exemple: reconnaître les mots-clés, variables, entiers.

Updater le dessin (code -¿ suite de lexemes)

Analyse syntaxique But: obtenir une structure plus facile à manipuler que le code initial.

Exemples : expressions arithmétiques, if, for, while

Updater le dessin (lexèmes -¿ arbre syntaxique)

Typage But: ajouter de la sémantique à l'arbre.

Exemple: typage

Updater le dessin (arbre -¿ arbre étiqueté)

2 Langages réguliers

2.1 Définition

Un *alphabet* est un ensemble (fini) de lettres. Ici, on fixe un alphabet $A = \{a, \ldots, z\}$. Un *mot* sur A est une suite de lettres de A (éventuellement vide, qu'on note ε). Un *langage* est un ensemble de mots.

2.2 Opérations sur les mots et les langages

La *concaténation* de deux mots u et v est le mot obtenu en lisant toutes les lettres de u puis celles de v.

Si L_1 et L_2 sont des langages, on définit :

- l'union $L_1 + L_2$ (composée des mots qui sont dans L_1 ou dans L_2)
- la concaténation $L_1 \cdot L_2$ (composée des concaténations d'un mot dans L_1 avec un mot dans L_2)
- l'itération L_1^* qui contient toutes les concaténations $u_1 \cdots u_n$ où les $u_i \in L_1$.

Exemples : A^*, $\{a\} \cdot \{b\}^*$, $\{a\}^* + \{ab\}^*$.

Pour représenter les langages on utilise des *expressions régulières*. Une expression régulière est:

- ε, qui représente le langage $\{\varepsilon\}$
- une lettre $a \in A$, qui représente $\{a\}$
- $E_1 + E_2$, qui représente l'union des langages représentés par E_1 et E_2
- $E_1 E_2$, qui représente la concaténation des langages représentés par E_1 et E_2
- E^*, qui représente l'itération du langage représenté par E

Exemples: $(a+b+c)^*$, ab^*, a^*+b^*.

Question : peut-on écrire une expression régulière qui reconnaît les mots-clés ? les entiers ?

Problème : comment savoir si un mot est dans un langage ?

2.3 Automates

Un *automate* est :

- un ensemble Q d'états
- un ensemble $I \subset Q$ d'états initiaux
- un ensemble $F \subset Q$ d'états finaux
- un ensemble δ de transitions (p, a, q) où $p, q \in Q$ et $a \in A$.

Dessin !

Un mot est *accepté* par un automate si c'est l'étiquette d'un chemin d'un état initial à un état final. Si $\mathcal{A}$ est un automate, on note $L(\mathcal{A})$ l'ensemble des mots qu'il accepte.

Théorème 1. *reconnu par expression régulière = reconnu par automate.*

Proof. Faire le sens expression =¿ automate (avec les epsilon-transitions + leur suppression) □

Un automate est *déterministe* si pour chaque état q et chaque lettre a, au plus une transition a sort de p.

Théorème 2. *Pour tout automate, il existe un automate déterministe qui reconnaît le même langage.*

Proof. Soit (Q, I, F, δ) un automate, on construit $(2^Q, \{I\}, \{S \in 2^Q, S \cap F \neq \varnothing\}, \delta^*)$, où $\delta^*(S, a) = \{q \in Q \mid \exists p \in S, (p, a, q) \in \delta\}$. □

2.4 Tout n'est pas régulier

On va montrer que le langage L des mots bien parenthésés sur l'alphabet $\{(,)\}$ n'est pas régulier. S'il l'est, il est reconnu par un automate $\mathcal{A}$, et notons N son nombre d'états. Soit $w_n = ({}^{N+1})^{N+1} \in L$.

Si on lit w_n dans $\mathcal{A}$, on doit tomber deux fois dans le même état en lisant les (. On peut alors supprimer les (lues entre ces deux occurrences, et on obtient un mot $({}^{p})^{N+1}$ avec $p < N+1$, qui est aussi accepté par $\mathcal{A}$, mais qui pourtant n'est pas dans L.

Théorème 3. *Si L est régulier, il existe N tel que tout mot de L de longueur au moins N peut se décomposer en xyz avec $|xy| \leqslant N$, $y \neq \varepsilon$ et $xy^kz \in L$ pour tout $k \geqslant 0$.*

3 Grammaires

3.1 Retour sur la compilation

Avec les langages réguliers, on peut reconnaître les lexèmes et faire de l'analyse lexicale.

Mais il y a des langages pas réguliers mais utiles (for imbriqués, commentaires,...). Comment on fait → besoin de règles "une instruction = while condition do instructions"

Une *grammaire* est :

- Un ensemble $\mathcal{T}$ de terminaux (les lexèmes)
- Un ensemble $\mathcal{N}$ de non-terminaux
- Un non-terminal S "de départ"
- Des règles de dérivation $N \to w$ où N est un non-terminal et w un mot sur $\mathcal{T} \cup \mathcal{N}$.

Le langage reconnu par une grammaire est l'ensemble des mots sur $\mathcal{T}$ en lesquels peut se dériver S.

Exemples : $S \to aSb|\varepsilon$, $S \to (S)S|\varepsilon$, $E \to (E+E)|E \times E|a$.

But : savoir si un mot peut être généré par la grammaire. Et si oui, en utilisant quelles règles ?

3.2 Analyse syntaxique

Il y a deux types d'analyse : descendante (on part de S et on essaye de le transformer en le mot) ou ascendante (on part du mot et on essaye de le transformer en S). Ici, on va faire de l'analyse descendante.

Prenons la grammaire $E \to E+T|T$, $T \to T \times F|F$, $F \to (E)|a$. Pour tester si w est dérivable à partir de E, ce qu'on aimerait, c'est quelque chose qui nous dise quelle règle a été appliquée à partir de E, en lisant le début de w.

Regardons quelles sont les premières lettres possibles d'un mot du langage. Si on part de F, on génère des mots commençant par (ou a. On note $premier(F) = \{(, a\}$. Si on part de T, on peut dériver F, donc aussi des mots commençant par (ou a et $premier(T) = \{(, a\}$. Et c'est pareil pour E. Mais on a un problème : si w commence par (, ça ne nous dit pas quelle règle entre $E \to E+T$ et $E \to T$ a été utilisée. C'est un problème de notre grammaire : elle n'est pas $LL(1)$, c'est-à-dire qu'on ne peut pas déterminer quelle règle a été utilisée en regardant seulement la première lettre du mot.

On va utiliser une autre grammaire, qui reconnaît le même langage, mais qui est $LL(1)$:

$$E \to TE' \qquad E' \to +TE'|\varepsilon \qquad T \to FT' \qquad T' \to \times FT'|\varepsilon \qquad F \to (E)|a$$

Ici, on a $premier(F) = \{(, a\}$, $premier(T') = \{\times, \varepsilon\}$, $premier(T) = \{(, a\}$, $premier(E') = \{+, \varepsilon\}$ et $premier(E) = \{(, a\}$.

Attention, il y a des règles $T' \to \varepsilon$ et $E' \to \varepsilon$, qui sont un peu embêtantes. En effet, on peut les appliquer sans générer de lettre dans le mot final. Pour savoir quand on peut les appliquer, il faut aussi calculer quelles lettres peuvent suivre un sous-mot dérivé depuis T' et E'.

Ici, on a $suivant(E) = \{), \$\}$ à cause de $F \to (E)$. Et donc $suivant(E') = \{), \$\}$ aussi à cause de $E \to TE'$. Ce qui peut suivre T, c'est ce qui peut commencer E' à cause de $E' \to +TE'$, mais aussi ce qui suit E' à cause de $E' \to \varepsilon$, donc $suivant(T) = \{+,), \$\}$.

En itérant, on trouve $suivant(T') = \{+,), \$\}$ et $suivant(F) = \{\times, +,), \$\}$.

Maintenant on peut faire une table à double entrée qui dit quelle règle appliquer pour dériver un non-terminal en un mot qui commence par une lettre donnée :

	$+$	$\times$	$($	$)$	a	$\$$
F			$F \to (E)$		$F \to a$	
T			$T \to FT'$		$T \to FT'$	
E			$E \to TE'$		$E \to TE'$	
T'	$T' \to \varepsilon$	$T' \to \times FT'$		$T' \to \varepsilon$		$T' \to \varepsilon$
E'	$E' \to +TE'$			$E' \to \varepsilon$		$E' \to \varepsilon$

Et on peut faire l'analyse de $a + a \times (a)$ pour voir.

Les règles de calcul :

- $premier(a) = \{a\}$
- Si $A \to \varepsilon$, $\varepsilon \in premier(A)$
- Si $X \to Y_1 \cdots Y_n$, et $\varepsilon \in premier(Y_1) \cap \cdots \cap premier(Y_{i-1})$ alors $premier(Y_i) \setminus \{\varepsilon\} \subset premier(X)$.
- Si $X \to Y_1 \cdots Y_n$, et $\varepsilon \in premier(Y_1) \cap \cdots \cap premier(Y_n)$ alors $\varepsilon \in premier(X)$.
- $\$ \in suivant(S)$
- Si $A \to \alpha B \beta$, alors $premier(\beta) \setminus \{\varepsilon\} \subset suivant(B)$
- Si $A \to \alpha B$ ou $A \to \alpha B \beta$ avec $\varepsilon \in premier(\beta)$, alors $suivant(A) \subset suivant(B)$

Et pour la table :

- On ajoute chaque règle $A \to \alpha$ dans les cases (A, x) pour chaque $x \in premier(A)$.
- Si $\varepsilon \in premier(\alpha)$, on ajoute $A \to \alpha$ aux cases (A, x) pour $x \in suivant(A)$.

3.3 Analyse ascendante

un jour peut-être ? en TD ?

4 Typage et unification

4.1 Intro

problématique : comment gérer "5432"+1 ?

- Erreur de compilation (caml) ?
- Erreur à l'exécution (python) ?
- 5433 (php) ?
- "54321" (java) ?
- BLURG (C) ?

Typage = donner un type à chaque expression + rejeter les programmes incohérents. Peut-être fait :

- à la compilation (C, Java, caml) après l'analyse syntaxique
- à l'exécution (python, php)

Qu'est-ce qu'on demande au programmeur ?

- Typer toutes les expressions (BERK)
- Typer ses variables (C, Java)
- Typer ses fonctions
- Rien (Ocaml)

4.2 Ce qu'on ne fait pas

Conversion de type : $1.2 + 42$?

- Rejeter (Ocaml) ?
- Convertir 42 en float (C, Java) ? $\rightarrow$ modifier AST

Polymorphisme : + additionne des entiers et des flottants, mais a le même nom.

4.3 Ce qu'on fait

Inférence totale : le programmeur peut ne rien annoter. On va trouver le type le plus général possible pour chaque expression (et annoter chaque noeud de l'arbre syntaxique).

EXEMPLE auto x; x=x+1; $f(x) = [x]$.

On va donc avoir des variables de types, et chaque type va être formé à partir de types de bases, de types variables, en utilisant des constructeurs (ex : list, tab, $\rightarrow$)

```
def f(x) : return [x]

def g(x,y) : return [x]+y

auto x,y;
[y,f(x)]
[x,f(x)]
```

4.4 Unification

But du jeu :

- "unifier" deux types = trouver comment remplacer les variables de type pour que les deux types deviennent les mêmes.
- le mieux possible = faire les substitutions les plus générales possible.

On cherche un unificateur, c'est-à-dire une liste de substitutions à effectuer. On va considérer nos types comme étant :

1. int, bool
2. une variable
3. $tab[T]$ pour représenter les tableaux dont les éléments ont le type T
4. $(T_1, \ldots, T_n) \to T$ pour représenter les fonctions

On initialise un ensemble S contenant la paire à unifier. Tant que S est non vide, on prend un élément $(a, b) \in S$.

- Si $a = b$, on retire (a, b) de S.
- Sinon, supposons que a est une variable de type. Si a apparaît dans b, on rejette, sinon on remplace tous les a par des b dans S et on ajoute $[a \to b]$ à l'unificateur.
- Sinon, ni a ni b ne sont des variables de type. Si $a = tab[a']$ et $b = tab[b']$, on ajoute (a', b') à S
- Si $a = (T_1, \ldots, T_n) \to T$ et $b = (T'_1, \ldots, T'_n) \to T'$, on ajoute (T, T') et tous les (T_i, T'_i) à S.
- Sinon, on rejette.

Exemple : $x \to (x \to z)$ et $tab[y] \to (z \to tab[int])$

4.5 En pratique

Quelques exemples :

- Affectation : $x =$ truc. Il faut vérifier que les types de x et truc sont bien compatibles (et éventuellement modifier le type connu de x en conséquence).
- Appel de fonction : $h(y) + 3$ où $h : y \to y[0]$. Il faut vérifier qu'il y a bien un typage de h ici qui peut renvoyer un entier (et on récupère au passage que y doit être un tableau d'entiers).
- Type de retour d'une fonction : si on a plusieurs return, il faut vérifier qu'ils renvoient bien des types unifiables pour pouvoir typer la fonction.

On va parcourir l'arbre et retenir un tableau de types.
Problèmes : portée de variables. Exemple

```
auto x;

while (blabla) {
    auto x = y+1;
    x = x + 3;
}

return x[0]+2;
```

On va en fait avoir une pile de tableaux : on empile un nouveau "contexte" quand on entre dans un bloc, et on le dépile (en stockant les résultats quelque part) quand on sort du bloc.

5 Génération de code

code linéaire + refaire le while
gérer les tableaux ?

- tableau = pointeur sur la longueur + 10 cases + un pointeur
- besoin d'un heapPointer pour savoir où on a de la mémoire libre
- accès $t[i]$ peuvent lancer segfault
- écriture $t[i] = x$ agrandissent
- initialisation sinon problèmes avec $t[2][3] = 4$

gérer les appels de fct ?

- empiler tout le contexte avant CALL + dépiler après sinon problèmes
- ça ne marche pas en code linéaire parce que absurdement gros → boulot pour le groupe 3 une fois qu'on a peu de registres

6 Allocation de registres

exemples motivation :

- un registre plus jamais utilisé
- un utilisé après un jump qui pourrait être partagé

Besoin d'une meilleure structure de données. Définition des blocs et du CFG
Exemple

```
x:=2;
y:=4;
x:=1;
if (y>x)
    z:=y;
else
    z:=y*y;
x:=z;
```

Variable vivante = variable qui *peut* être utilisée plus tard.
Disclaimer:

- overapprox (parce que analyse statique)
- on s'est mis à l'échelle du bloc, mais on peut aussi se placer à l'échelle de l'instruction

Vraie définition : v vivante à la fin de B si on a un chemin qui part de B et qui finit à une utilisation de v sans l'écraser.
Comment ça se calcule ?

- variable tuée par un bloc = variable dont le contenu est écrasé
- variable générée par un bloc = variable dont on a besoin de la valeur (avant qu'elle se fasse éventuellement écraser)
- LV exit = variables vivantes à la fin d'un bloc
- LV entry = variables vivantes au début d'un bloc

Équations de point fixe : exit = union des entry des successeurs, entry = exit - kill + gen.
Code inutile : on peut supprimer tous les a :=truc si a pas dans LV exit.
Exemple Matthieu 4.3
Deux registres ne peuvent pas être fusionnés s'ils sont vivants dans un même bloc. Graphe de conflit + exemple

```
LD R1 R0
SUBi R0 R0 1
LD R2 R0
SUBi R0 R0 1
LD R3 R0
ADD R4 R3 R2
ADDi R5 R4 0
ADD R6 R1 R5
```

But : associer un registre à chaque noeud du graphe, minimiser le total, sans mettre deux sommets en conflit dans le même registre → coloration.

Comment on trouve une coloration opti : NP-dur. Glouton ? marche pas toujours et peut être loin de l'optimal.

Gérer l'appel de fonction + faire un CFG par fonction

On fait quoi de ça ? Moins de 32 registres → youpi. Sinon, il faut utiliser la mémoire. Idée débile : LD/ST à chaque utilisation.