Langage de Programmation et Compilation

Jean-Cristophe Filliâtre

27 octobre 2023

Table des matières

Première partie

Cours 1 29/09

Table des matières

1 Introduction à la Compilation

Maîtriser les mécanismes de la compilation, transformation d'un langage dans un autre. Comprendre les aspects des langages de programmation.

1.1 Un Compilateur

Un compilateur est un traducteur d'un langage source vers un langage cible. Ici le langage cible sera l'asembleur.
Tous les langages ne sont pas compilés à l'avance, certains sont interprétés, transpilés puis interprétés, compilés à la volée, transpilés puis compilés... Un compilateur prend un programme P et le traduit en un programme Q de sorte que : $\forall P, \exists Q, \forall x,\ P(x) = Q(x)$. Un interpréteur effectue un travail simple mais le refait à chaque entrée, et donc est moins efficace.
Exemple : le langage *lilypond* va compiler un code source en fichier .pdf.

1.2 Le Bon et le Mauvais Compilateur

On juge un compilateur à :

1. Sa correction
2. L'efficacité du code qu'il produit
3. Son efficacité en tant que programme

> « Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free ! Thus, the most important objective in writing a compiler is that it is correct »- *Dragon Book, 2006*

1.3 Le Travail d'un Compilateur

Le travail d'un compilateur se compose :

— d'une phase d'analyse qui :
   1. reconnaît le programme à traduire et sa signification
   2. signale les erreurs et peut donc échouer

— d'une phase de synthèse qui :
   1. produit du langage cible
   2. utilise de nombreux langages intermédiaires
   3. n'échoue pas

Processus : source → analyse lexicale → suite de lexèmes (tokens) → analyse syntaxique → Arbre de syntaxe abstraite → analyse sémantique → syntaxe abstraite + table des symboles → production de code → langage assembleur → assembleur → langage machine → éditeur de liens → exécutable.

2 L'assembleur

2.1 Arithmétique des ordinateurs

On représente les entiers sur n bits numérotés de droite à gauche. Typiquement, n vaut 8, 16, 32 ou 64. On peut représenter des entiers non signés jusqu'à $2^n - 1$. On peut représenter les entiers en définissant b_{n-1} comme un bit de signe, on peut alors représenter $\left[-2^{n-1}, 2^{n-1} - 1\right]$. La valeur d'une suite de bits est alors : $-b_{n-1}2^{n-1} + \sum_{k=0}^{n-2} b_k 2^k$. On ne peut pas savoir si un entier est signé sans le contexte.
La machine fournit des opérations logiques (bit à bit), de décalage (ajout de bits 0 de poids fort, 0 de poids faible ou réplication du bit de signe pour interpréter une division), d'arithmétique (addition, soustraction, multiplication).

2.2 Architecture

Un ordinateur contient :

— Une unité de calcul (CPU) qui contient un petit nombre de registres et des capactités de calcul

— Une mémoire vive (RAM), composée d'un très grand nombre d'octets (8 bits), et des données et des instructions, indifférenciables sans contexte.

L'accès à la mémoire coûte cher : à 1B instructions/s, la lumière ne parcourt que 30cm entre deux instructions.
En réalité, il y a plusieurs (co)processeurs, des mémoires cache, une virtualisation de la mémoire...
Principe d'exécution : un registre (%rip) contient l'adresse de l'instruction, on lit (fetch) un ou plusieurs octects dans la mémoire, on interprète ces bits (decode), on exécute l'instruction (exectue), on modifie (%rip) pour l'instruction suivante. En réalité, on a des pipelines qui branchent plusieurs instructions en parallèle, et on essaie de prédire les sauts conditionnels.
Deux grandes familles d'Architectures : CISC (complex instruction set), qui permet beaucoup d'instructions différentes mais avec assez peu de registres, et RISC (Reduced Instruction Set) avec peu d'instruction effectuées très régulièrements et avec beaucoup de registres. Ici, on utilisera l'architecture *x86-64*.

2.3 L'architecture `x86-64`

Extension 64 bits d'une famille d'architectures compatibles Intel par AMD adoptée par Intel. Architecture à 16 registres, avec adressage sur 48 bits au moins et de nombreux modes d'adressage. On ne programme pas en langage machine mais en assembleur, langage symbolique avec allocation de données globales, qui est transformé en langage machine par un assembleur qui est en réalité un compilateur. On utilise l'assembleur GNU avec la syntaxe AT&T (la syntaxe Intel existe aussi).

2.4 L'assembleur `x86-64`

Pour assembler un programme assembleur, appeler `as -o file.o` puis appeler l'édition de lien avec `gcc -no-pie file.s -o exec-name`. On peut débuguer en ajoutant l'option `-g`. La machine est petite boutiste (little-endian) si elle stocke les valeurs dans la RAM en commençant par le bit de poids faible, gros boutiste (big-endian) pour le poids fort.
Commandes : Dans cette liste, $\%(r)$ désigne l'adresse mémoire stockée dans r

— `movq $a %b` permet de mettre la valeur a dans le registre b

— `movq %a %b` permet de copier le registre a dans le registre b

— `movq $label %b` permet de changer l'adresse de l'étiquette dans le registre b

— `addq %a %b` permet d'additionner les registres a et b.

— `incq %r` permet d'incrémenter le registre r, de même pour `decq`.

— `negq %r` permet de modifier la valeur de r en sa négation

— `notq %r` permet de modifier la valeur de r en sa négation logique.

— `orq %r1 %r2` (resp. `andq` et `xorq`) permet d'affecter à $r2$, $or(r1, r2)$ (resp. and, xor)

— `salq $n %r`/`salq %cl %r` décale la valeur de r de n (ou $\%cl$) zéros à gauche.

— `sarq` est le décalage à droite arithmétique, `shrq` le décalage à droite logique.

— Le suffixe `q` désigne une opération sur 64 bits. `b` désigne 1 octet, `w` désigne 2 octets, `l` désigne 4 octets. Il faut préciser les deux extensions si celles-ci diffèrent.

— `jmp label` permet de jump à une étiquette.

La plupart des opérations fositionnent des drapeaux selon leur résultat.
Certaines instructions : `j(suffixe)` (jump), `set(suffixe)` et `cmov(suffixe)`(move) permettent de tester des drapeaux et d'effectuer une opération selon leur valeur.

On ne sait pas combien il y a d'instructions en `x86-64`.

2.5 Le Défi de la Compilation

C'est de traduire un programme d'un langage de haut niveau vers ce jeu d'instruction.
Constat : les appels de fonctions peuvent être arbitrairement imbriqués et les registres ne suffisent pas ⇒ on crée alors une pile car les fonctions procèdent majoritairement selon un mode LIFO.
La pile est stockée tout en haut, et croît dans le sens des adresses décroissantes, `%rsp` poînte sur le sommet de la pile. Les données dynamiques sont allouées sur le tas, en bas de la zone de données.
Chaque programme a l'illusion d'avoir toute la mémoire pour lui tout seul, illusion créée par l'OS.
En assembleur on a des facilités d'utilisation de la pile :

— `pushq $a` push a dans la pile

— `popq %rdi` dépile

Lorsque f (caller) appelle une fonction g (callee), on ne peut pas juste `jmp g`. On utilise `call g` puis une fois que c'est terminé `ret`.
Mézalor tout registre utilisé par g sera perdu par f. On s'accorde alors sur des **conventions d'appel**. Des arguments sont passés dans certains registres, puis sur la pile, la valeur de retour est passéee dans `%rax`. Certains registres sont *callee-saved* i.e. l'appelé doit les sauvegarder pour qu'elle survivent aux appels. Les autres registres sont dit *caller-saved* et ne vont pas survivre aux appels.
Il faut également qu'en entrée de fonction `%rsp + 8` doît être multiple de 16, sinon des fonctions peuvent planter.
Il y a quatres temps dans un appel :

1. Pour l'appelant, juste avant l'appel :
2. Pour l'appelé, au début de l'appel :
   (a) Sauvegarde `%rbp` puis le positionne.
   (b) Alloue son tableau d'activation.
   (c) Sauvegarde les registres *callee-saved*.
3. Pour l'appelé, à la fin de l'appel :
   (a) Placer le résultat dans `%rax`
   (b) Restaure les registres sauvegardés
   (c) Dépile sont tableau d'activation
   (d) Exécute `ret`
4. Pour l'appelant, juste après l'appel :
   (a) Dépile les éventuels arguments
   (b) Restaure les registres *caller-saved*

Deuxième partie

Cours 2 : 6/10

Table des matières

Introduction

La signification des programmes est définie souvent de manière informelle, en langue naturelle, e.g. le langage Java.

3 Sémantique Formelle

La sémantique formelle caractérise mathématiquement les calculs décrits par un programme. C'est utile pour la réalisation d'outils (interprètes, compilateurs), et nécessaire aux raisonnements sur les programmes.

3.1 Syntaxe Abstraite

On ne peut pas manipuler un programme en tant qu'object syntaxique, on préfère utiliser la syntaxe abstraite (se déduit lors de la compilation à l'analyse syntaxique et sémantique.). On construit un arbre de syntaxe abstraite pour comprendre.
On définit la syntaxe abstraite par une grammaire. En OCamL, on réalise la syntaxe abstraite par des types construits. Il n'y a pas de parenthèses dans la syntaxe abstraite. On appelle sucre syntaxique toute construction de la syntaxe concrète qui n'existe pas dans la syntaxe abstraite.

C'est sur la syntaxe abstraite qu'on va définir la sémantique.

3.2 Sémantiques Useless

3.2.1 Sémantique Axiomatique - Logique de Floyd-Hoare

Tony Hoare, An axiomatic basis for computer programming, 1969, article le plus cité de l'histoire de l'informatique.
On caractérise les programmes par l'intermédiaire des propriétés satsfaites par les variables. On introduit le triplet $\{P\}\ i\ \{Q\}$ qui signifie, si P est vraie avant l'instruction i, après, Q sera vraie

3.2.2 Sémantique Dénotationelle

A chaque expression e, on associe sa définition $\|e\|$ qui est un objet représentant le calcul désigné par e. On définit cet objet récursivement.

3.2.3 Sémantique Par Traduction

On définit la sémantique d'un langage en le traduisant vers un langage dont la sémantique est connue.

3.3 Sémantique Opérationnelle

La sémantique opérationnelle décrit l'enchaînement des calculs élémentaires qui mènent de l'expression à son résultat.
Il y a deux formes de sémantique opérationelle :

1. La sémantique naturelle (big-steps semantics) : $e \twoheadrightarrow v$
2. La sémantique par réduction (small steps semantics) $e \to e_1 \to e_2 \to \ldots \to v$

On applique ça à Mini-ML (qui est Turing-Complet).

3.3.1 Sémantique Opérationnelle à Grand Pas

On cherche à définir $e \twoheadrightarrow v$. On définit les valeurs parmi les constantes, les primitives non appliquées, les fonctions et les paires.
Une relation peut être définie comme la plus petite relation satisfaisant un ensemble d'axiomes notés $\overline{P}$ et des règles d'inférences (implications). On définit Pair(n) par $\overline{Pair(0)}$ et $\frac{Pair(n)}{Pair(n+2)}$. La plus petitre relation qui vérifie ces deux propriétés coïncide avec la propriété « n est un entier naturel pair ».
Une dérivation est un arbre dont les noeuds correspondent aux règles et les feuilles aux axiomes (les arbres croissent vers le haut). L'ensemble des dérivations possibles caractérise exactement la plus petite relation satisfaisant les règles d'inférence.

Définition 3.3.1. *On définit les variables libres d'une expression e, noté $fv(e)$ par récurrence sur e avec :*

$$\begin{aligned} fv(x) &= \{x\} \\ fv(c) &= \varnothing \\ fv(op) &= \varnothing \\ fv(fun\ x \to e) &= fv(e) \setminus \{x\} \\ fv(e_1 e_2) &= fv(e_1) \cup fv(e_2) \\ fv((e_1, e_2)) &= fv(e_1) \cup fv(e_2) \\ fv(let\ x = e_1\ in\ e_2) &= fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\ fv(x) &= \{x\} \end{aligned}$$

On défnit la substitution de toute occurence libre de x dans e par v définie par :

$$\begin{aligned} x[x \leftarrow v] &= v \\ y[x \leftarrow v] &= y\ si\ y \neq x \end{aligned}$$

On fait le choix d'une stratégie d'appel par valeur, i.e. l'argument est complètement évalué avant l'appel.
On a ici comme axiomes : $\overline{c \twoheadrightarrow c}$, $\overline{op \twoheadrightarrow op}$, $\overline{(\text{fun } x \to e) \twoheadrightarrow (\text{fun } x \to e)}$ et comme règles d'inférences :

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2 \twoheadrightarrow v_2}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)} \qquad \frac{e_1 \twoheadrightarrow v_1 \quad e_2[x \leftarrow v_1] \twoheadrightarrow v}{let\ x = e_1 \text{ in } e_2 \twoheadrightarrow v}$$

et

$$\frac{e_1 \twoheadrightarrow (\text{fun } x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

On ajoute ensuite des règles pour les primitives, dépendant de la forme de chacune, e.g. :

$$\frac{e_1 \twoheadrightarrow +\quad e_2 \twoheadrightarrow (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \twoheadrightarrow n}$$

Partant, on peut montrer qu'un programme s'évalue en une valeur en écrivant l'arbre de dérivation de celui-ci.

Remarque 3.3.0.1. *Il existe des expressions sans valeur : $e = 12$ par exemple.*

On peut établir une propriété d'une relation définie par un ensemble de règles d'inférence, en raisonnant par induction sur la dérivation. Cela signifie par récurrence structurelle.

Proposition 3.3.1. *Si $e \twoheadrightarrow v$, alors v est valeur. De plus si e est close alors v l'est également.*

Démonstration. Par induction : $\frac{e_1 \twoheadrightarrow (\text{fun } x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1 e_2 \twoheadrightarrow v}$. ■

Proposition 3.3.2. *Si $e \twoheadrightarrow v$ et $e \twoheadrightarrow v'$, alors $v = v'$.*

Démonstration. Par induction. ■

Remarque 3.3.0.2. *On a donc définit une fonction plus qu'une relation.*

3.3.2 Sémantique à Petits Pas

La sémantique opérationnelle à petits pas remédie aux problèmes de programmes qui ne terminent pas, en introduisant une notion d'étape élémentaire de calcul $e_1 \to e_2$
On commence par définir une relation $\to^\varepsilon$ correspondant à une réduction en tête, au sommet de l'expression, par exemple : (fun $x \to e$) $v \to^\varepsilon e[x \leftarrow v]$ On se donne également des règles pour les primitives. On réduit en profondeur en introduisant la règle d'inférence : $\frac{e_1 \to^\varepsilon e_2}{E(e_1) \to E(e_2)}$ où E est un contexte défini par la grammaire suivante :

$$\begin{array}{rcl} E & ::= & \square \\ & | & Ee \\ & | & vE \\ & | & \text{let } x = E \text{ in } e \\ & | & (E, e) \\ & | & (v, E) \end{array}$$

Un Contexte est un terme à trou où $\square$ représente le trou. $E(e)$ dénote le contexte E dans lequel $\square$ a été remplacé par e. La règle d'inférence permet donc d'évaluer une sous-expression. Tels que définis, les contextes impliquent ici une évaluation en appel par valeur et de gauche à droite.
On note $\to^*$ la cloture réflexive et transitive de $\to$.

Définition 3.3.2. *On appelle forme normale toute expression e telle qu'il n'existe pas e' telle que : $e \to e'$*

3.3.3 Equivalence des Sémantiques

Théorème 3.3.1. *Les deux sémantiques opérationnelles sont équivalentes pour les expressions dont l'évaluation termine sur une valeur i.e. :*

$$e \twoheadrightarrow v \Leftrightarrow e \to^* v$$

Démonstration.

Lemme 3.3.2 (Passage au contexte des réductions)**.** *Supposons $e \to e'$, alors :*

1. $ee_2 \to e'e_2$
2. $ve \to ve'$
3. *let* $x = e$ *in* $e_2 \to$ *let* $x = e'$ *in* e_2

— ($\Rightarrow$) On procède par induction sur la dérivation.

— ($\Leftarrow$) :

Lemme 3.3.3 (Evaluation des Valeurs)**.** *On a $v \twoheadrightarrow v$.*

Lemme 3.3.4. *Si $e \to e'$ et $e' \twoheadrightarrow v$ alors $e \twoheadrightarrow v$.*

Démonstration. On commence par les réductions de tête, puis on procède par induction aux applications de contexte. ■

On a alors, par récurrence sur le nombre de pas, l'implication souhaitée.

■

3.3.4 Langages Impératifs

Pour un langage impératif les sémantiques ci-dessus sont insuffisantes. On associe alors typiquement un état S à l'expression évaluée. L'état peut être décomposé en plusieurs éléments pour modéliser par exemple une pile (des variables locales), des tas...

4 Interprète

On peut programmer un interprète en suivant les règles de la sémantique naturelle. On se donne un type pour la syntaxe abstraite des expressions et on définit une fonction correspondant à la relation $\twoheadrightarrow$
Un interprète renvoie la (ou les) valeur(s) d'une expression, souvent récursivement.
On peut éviter l'opération de substitution, en interprétant l'expression e à l'aide d'un environnement donnant la valeur courante de chaque variable (un dictionnaire). Ceci pose problème car le résultat de let $x = 1$ in fun $y \to +(x, y)$ est une fonction qui doit « mémoriser »que $x = 1$.
On utilise alors le module Map pour les environnements (c'est une *fermeture*). On représente alors la valeur d'une fonction avec son environnement.
Pour un interprète de la sémantique à petits pas, il vaut mieux utiliser un *zipper* que de recalculer le contexte tout le temps.

Troisième partie
Cours 3 : 13/10

Table des matières

Introduction

L'objectif est de partir du code source, une suite de caractère, pour obtenir une suite de lexèmes plus compréhensible et simple à analyse syntaxiquement

5 Blancs

Les blancs (espace, retour chariot, tabulation) jouent un rôle dans l'analyse lexicale, car ils permettent de séparer deux lexèmes. De nombreux blancs sont inutiles e.g. x + 1, et seront ignorés. Les conventions diffèrent selon les langages et certains des caractères blancs peuvent être significatifs, par exemple l'indentation en python ou en Haskell, ou les retours chariots transformés en points-virgules comme en python. Les commentaires jouent le rôle de blancs.

6 Outils pour l'analyse lexicale

On va utiliser des expressions régulières pour décrire les lexèmes et des automates finis pour les reconnaître. On exploite en particulier la capacité à construire un automate partant d'une ou plusieurs expressions régulières.

6.1 Expressions Régulières et Automates Finis [1]

6.1.1 Expressions Régulières

Définition 6.1.1 (Syntaxe)**.** *On définit la syntaxe des expressions régulières :*

$$\begin{aligned} r = &| \quad \varnothing \\ &| \quad \varepsilon \\ &| a \in \Sigma \\ &| \quad r \cdot r \\ &| r + r \\ &| \quad r\star \end{aligned}$$

Définition 6.1.2 (Sémantique)**.** *On définit alors la sémantique basée sur cette syntaxe par les langages rationnels :*

$$\begin{aligned} L(\varnothing) &= \varnothing \\ L(a) &= \{a\} \\ L(r_1 r_2) &= L(r_1) \cdot L(r_2) \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r\star) &= \bigcup_{n\in\mathbb{N}} L^n(r) \end{aligned}$$

Pour les constantes flottantes de CamL on a par exemple, si $d = 0|1|\ldots|9$:

$$d\ d\star\,(.d\star \mid (\varepsilon \mid .d\star)(e \mid E)(\varepsilon \mid + \mid -)\ d\ d\star)$$

On peut alors écrire un algorithme pour savoir si une suite de caractère appartient à une regexp.

Définition 6.1.3 (Dérivée de Brzozowski)**.** *On pose, pour r une regexp et $c \in \Sigma$: $\delta(r, c) = \{w \mid cw \in L(r)\}$*

6.1.2 Automates Finis

Définition 6.1.4 (Syntaxe)**.** *Un automate fini est un quintuplet (Q, Σ, I, F, T) où :*

- *Q est un ensemble fini d'états*
- *Σ est un ensemble fini appelé alphabets*
- *$I \subseteq Q$ est un ensemble d'états initiaux*
- *$F \subseteq Q$ est un ensemble d'états finaux*
- *$T \subseteq Q \times \Sigma \times Q$ est un ensemble de transitions*

Théorème 6.1.1 (De Kleene)**.** *Les expressions régulières et les automates finis définissent les mêmes langages.*

Démonstration. Voir Cours de LFCC ■

1. Voir Cours de LFCC

6.2 Analyseur Lexical

6.2.1 Principe

Un analyseur lexical est un automate fini pour la réunion de toutes les expressions régulières définissant les lexèmes. Le fonctionnement de l'analyseur lexical est différent de la simple reconnaissance d'un mot par un automate car :

— Il faut décomposer un mot (le source) en une suite de mots reconnus

— Il peut y avoir des ambiguïtés

— Il faut construire les lex'emes (les états finaux contiennent des actions)

Ambiguïtés

Le mot funx est reconnu par l'expression régulière des identificateurs mais contient un préfixe reconnu par une autre expression régulière (fun) : ⇒ On choisit de reconnaître le lexème le plus long possible.
Le mot fun est reconnu par la regexp du mot clef fun mais aussi par celle des identificateurs : ⇒ On classe les lexèmes par ordre de priorité.

Retour en arrière

Un analyseur va échouer sur l'entrée abc avec les trois regexp a, ab, bc. L'analyseur lexical doit donc mémoriser le dernier état final rencontré, le cas échéant.

Lorqu'il n'y a plus de transition possible dans l'automate, de deux choses l'une :

— Soit aucun état final mémorisé : échec de l'analyse lexicale

— Soit on a lu le préfixe wv de l'entrée, avec w le lexème reconnu par le dernier état final recontré : on renvoie w et l'analyse lexicale redémarre avec v préfixé au reste de l'entrée

En pratique, on va renvoyer dans l'analyseur lexical une fonction de calcul du prochain lexème, puisque l'analyse lexicale est faite pour l'analyse syntaxique, cf. IV

6.2.2 Construction

L'automate de Thompson

On construit par induction un automate compatible.

L'automate de Berry-Sethi

On met en correspondance les lettres d'un mot reconnu et celles apparaissant dans la regexp : On distingue les différentes lettres de la regexp puis on construit un automate dont les états sont des ensembles de lettres. Pour construire les transitions de s_1 à s_2, on détermine les lettres qui peuvent apparaître après une autre dans un mot reconnu : follow. Pour calculer follow, on a besoin de savoir calculer les premières et dernières lettres d'un mot reconnu (first et last). On a alors besoin d'une dernière notion : null, est-ce que ε appartient au langage reconnu. On obtient :

$\text{null}(\varnothing)$	$=$	false
$\text{null}(\varepsilon)$	$=$	true
$\text{null}(a)$	$=$	false
$\text{null}(r_1 r_2)$	$=$	$\text{null}(r_1) \wedge \text{null}(r_2)$
$\text{null}(r_1 \mid r_2)$	$=$	$\text{null}(r_1) \vee \text{null}(r_2)$
$\text{null}(r\star)$	$=$	true
On en déduit :		
$\text{first}(\varnothing)$	$=$	$\varnothing$
$\text{first}(\varepsilon)$	$=$	$\varnothing$
$\text{first}(a)$	$=$	$\{a\}$
$\text{first}(r_1 r_2)$	$=$	$\text{first}(r_1) \cup \text{first}(r_2)$ si $\text{null}(r_1)$
	$=$	$\text{first}(r_1)$ sinon
$\text{first}(r_1 \mid r_2)$	$=$	$\text{first}(r_1) \cup \text{first}(r_2)$
$\text{first}(r\star)$	$=$	$\text{first}(r)$
On définit last de même		
$\text{follow}(c, \varnothing)$	$=$	$\varnothing$
$\text{follow}(c, \varepsilon)$	$=$	$\varnothing$
$\text{follow}(c, a)$	$=$	$\varnothing$
$\text{follow}(c, r_1 r_2)$	$=$	$\text{follow}(c, r_1) \cup \text{follow}(c, r_2) \cup \text{first}(r_2)$ si $c \in \text{last}(r_1)$
	$=$	$\text{follow}(c, r_1) \cup \text{follow}(c, r_2)$ sinon
$\text{follow}(c, r_1 \mid r_2)$	$=$	$\text{follow}(c, r_1) \cup \text{follow}(c, r_2)$
$\text{follow}(c, r\star)$	$=$	$\text{follow}(c, r) \cup \text{first}(r)$ si $c \in \text{last}(r)$
	$=$	$\text{follow}(c, r)$ sinon

On construit alors l'automate reconnaissant r en ajoutant # à la fin de r :

1. L'état initial est l'ensemble $\text{first}(r\#)$.
2. Tant qu'il existe un état s dont on doit calculer les transitions, pour chaque c de l'alphabet, on pose s' l'état $\bigcup_{c_i \in s} \text{follow}(c_i, r\#)$
3. Les états acceptants sont ceux contenant #

7 L'outil ocamllex

7.1 Analyseur Lexical en ocamllex

Un fichier ocamllex porte le suffixe .mll et a la forme suivante :

— Code OCamL arbitraire

— rule f_1 = parse | regexp1 {action 1}

— ...

— Code OcamL arbitraire

A la compilation par ocamllex file.mll, on construit un fichier OCamL contenant des fonctions de types Lexing.lexbuf → type où le type de sortie dépend de l'action dans la fonction.

Les regexp en ocamllex s'écrivent sous la forme :

_	n'importe quel caractère
'a'	le caractère 'a'
"foobar"	la chaîne "foobar" (en particulier $\varepsilon =$ "")
[caractères]	ensemble de caractères
[^caractères]	complémentaire d'un ensemble de caractères
$r_1 \mid r_2$	alternative
$r_1 r_2$	concaténation
$r*$	étoile
$r+$	une ou plusieurs occurences de r, i.e. $rr\star$
$r?$	au plus une occurence de r
eof	fin du fichier

Pour remplacer la reconnaissance du lexème le plus long par celui le plus court, remplacer parse par shortest.
A longueur égale, c'est la règle qui apparaît en premier qui l'emporte.
On peut nommer la chaîne reconnue, ou des sous-chaînes reconnues par des sous-experssions régulières, à l'aide de la construction as.
On peut dans une action, rappeler récursivement l'analyseur lexical ou l'un des autres analyseurs simultanément définis. Le tampon d'analyse lexical doit être passé en argument, il est contenu dans une variable appelée lexbuf. Ceci est utile pour séparer les blancs, ou reconnaître les commentaires, même imbriqués.

Par défaut ocamllex contruit l'automate dans une table interprétée à l'exécution, mais l'option -ml construit du code CamL pur.

7.2 Efficacité

Pour des raisons de stockage, et même en utilisant une table, l'automate peut prendre beaucoup de place. Il est donc préférable d'utiliser une seule expression régulière pour les identificateurs et les mots-clefs, puis de les séparer ensuite grâce à une table des mots-clefs.
On peut de même ne stocker que les caractères en minuscule pour être insensible à la casse.

7.3 D'autres utilisations ocamllex

On peut utiliser ocamllex pour :

1. Réunir plusieurs lignes vides consécutives en une seule
2. Compter les occurences d'un mot dans un texte
3. Un petit traducteur OCamL vers HTML pour embellir le source mis en ligne (mettre les mots-clefs en vert, les commentaires en rouge, numéroter les lignes ...), le tout en moins de 100 lignes de code.

Quatrième partie
Cours 4 : 20/10

8 Analyse Syntaxique

L'objectif de l'analyse syntaxique est, à partir d'une suite de lexèmes, de construire la syntaxe abstraite, qu'on représente sous forme d'arbre. En particulier, l'analyse syntaxique doit détecter les erreurs de syntaxe et les localiser précisément, les identifier (parenthèse non fermée, etc...) voire reprendre l'analyse pour découvrir de nouvelles erreurs. On va utiliser : une grammaire non contextuelle pour décrire la syntaxe et un automate à pile pour la reconnaître.

9 Grammaires

Définition 9.0.1. *Une grammaire non contextuelle est un quadruplet (N, T, S, R) où :*

- *N est un ensemble fini de symboles non terminaux.*
- *T est un ensemble fini de symboles terminaux.*
- *$S \in N$ est le symbole de départ (axiome).*
- *$R \subseteq N \times (N \cup T)^*$ est un ensemble fini de règles de production.*

On note les règles de dérivation sous la forme :

$$\begin{aligned} E \quad &\to E + E \\ &| E * E \\ &| (E) \\ &| \text{int} \end{aligned}$$

Définition 9.0.2. *Un mot $u \in (N \cup T)^*$ se dérive en un mot $v \in (N \cup T)^*$ et on note $u \to v$ s'il existe une décomposition $u = u_1 X u_2$ avec $X \in N$, $X \to \beta \in R$ et $v = u_1 \beta u_2$. On appelle dérivation gauche le cas où u_1 ne contient pas de mots terminaux. On appelle langage défini par G l'ensemble des mots de T^* dérivés de l'axiome.*

Définition 9.0.3. *Un arbre de dériviation est un arbre dont les noeuds sont étiquetés par des symboles de la grammaire, de la manière suivante :*

- *La racine est l'axiome*
- *Tout noeud interne X non terminal dont les fils sont étiquetés par $\beta \in (N \cup T)^*$ avec $X \to \beta$ une règle de la dérivation.*

Pour un arbre de dérivation dont les feuilles forment le mot w dans l'arbre infixe, on a $S \to^\star w$. Inversement, si $S \to^\star w$, on a un arbre de dérivation dont les feuilles dans l'arbre infixe forment w.

Définition 9.0.4. *Une grammaire est ambigüe si au moins un mot admet au moins deux arbres de dérivation. Déterminer si une grammaire est ambigüe ou non n'est pas décidable.*

La grammaire précédente est ambigüe, comment interpréter $\text{int} + \text{int} * \text{int}$

On va utiliser des critères décidables suffisants pour garantir qu'une grammaire est non ambigüe, et pour lesquels on sait en outre décider l'appartenance au langage efficacement.

10 Analyse ascendante

On va ici lire l'entrée de gauche à droite (pas toujours le cas, cf. CYK) puis reconnaître des membres droits de productions pour construire l'arbre de dérvation de bas en haut : bottom-up parsing.

10.1 Fonctionnement

On va manipuler une pile qui est un mot de $(N \cup T)^*$. A chaque instant, on a deux actions possibles :

- Une opération de lecture (*shift*) : on lit un terminal de l'entrée et on l'empile.
- Une opération de réduction (*reduce*) : on reconnaît en sommet de pile le membre droit β d'une production $X \to \beta$ et on remplace β par X en sommet de pile.

Comment prendre la décision lecture / réduction ? On se sert d'un automate fini et on examine les k premiers lexèmes de l'entrée, c'est l'analyse LR(k), pour « Left to right scanning, Righmost derivation ». En pratique, $k = 1$.

10.2 Analyse LR

La pile est de la forme $s_0x_1s_1x_2 \dots x_ns_n$ où s_i est un état de l'automate et $x_i \in T \cup N$ comme dans un automate. Soit a le premier lexème de l'entrée. Une table indexée par s_n et a nous indique l'action à effectuer :

- Si c'est un succès ou un échec, on s'arrête.
- Si c'est une lecture, on empile a et l'état s résultat de la transition $s_n \rightarrow^a s$ dans l'automate.
- Si c'est une réduction $X \rightarrow \alpha$, avec α de longueur p, on doit trouver α en sommet de pile :

$$s_0x_1 \dots x_{n-p}s_{n-p} \mid \alpha_1 s_{n-p+1} \dots \alpha_p s_n$$

  On dépile alors α et on empile Xs où $s_{n-p} \rightarrow^X s$ dans l'automate, i.e. $s_0x_1 \dots x_{n-p}s_{n-p}Xs$

  En pratique on ne travaille pas avec l'automate mais avec deux tables :

- Une table d'actions ayant pour lignes les états et pour colonnes les terminaux. La case action(s, a) indique :
  - shift s' pour une lecture et un nouvel état $s^{'}$
  - reduce X $\rightarrow \alpha$ pour une réduction
  - un succès
  - un échec

  Une table de déplacements ayant pour lignes les états et pour colonnes les non terminaux. La case goto(s, X) indique l'état résultat d'une réduction de X.

On ajoute aussi un lexème spécial # qui désigne la fin de l'entrée. On peut le voir comme l'ajout d'un nouveau non terminal S qui devient l'axiome et d'une règle $S \rightarrow E\#$. Pour créer ces tables, on utilise la famille de yacc (*Yet Another Compiler Compiler*)

11 L'outil Menhir

Menhir transforme une grammaire en un analyseur OCamL de type $LR(1)$. Chaque production de la grammaire est accompagnée d'une action sémantique, i.e. du code OCamL construisant une valeur sémantique (typiquement un arbre de syntaxe abstraite). Menhir s'utilise conjointement avec un analyseur lexical (typiquement ocamllex).
Un fichier Menhir a le suffixe .mly et a la forme suivante :

- %code OCamL arbitraire %
- déclaration des lexèmes
- %%
- non-terminal-1 : | production { action } | production { action } ;
- non-terminal-2 : | production { action }
- %%
- code OCamL arbitraire

En cas de conflits, Menhir produit deux fichiers pour expliquer d'où les conflits viennent. On peut les résoudre est d'indiquer comment choisir entre lecture et réduction. On peut donner des priorités aux lexèmes et aux productions, en fonction des règles d'associativité. Si la priorité de la production est supérieure à celle du lexème à lire, la réduction est favorisée. En cas d'égalité, l'associativité est consultée : un lexème associatif à gauche favorise la réduction et un lexème associatif à droite favorise la lecture. Par ailleurs, pour empêcher l'associativité, on peut aussi l'indiquer à Menhir, et éviter le *dangling else*.

Pour que les phases suivantes de l'analyse (typiquement le typage) puissent localiser les messages d'erreur, il convient de conserver une information de localisation dans l'arbre de syntaxe abstraite. Menhir fournit cette information dans $*startpos* et $*endpos*, deux valeurs du type Lexing.position. Cette information lui a été transmise par l'analysur lexical. (ocamllex ne maintient automatiquement que la position absolue dans le fichier, il faut appeler Lexing.new_line pour chaque retour chariot.)

12 Derrère l'outil Menhir

12.1 First, Null, Follow

Définition 12.1.1 (Null). *Soit $\alpha \in (T \cup N)^*$. Null(α) est vrai si et seulement si on peut dériver ε à partir de α.*

Définition 12.1.2 (First). *Soit $\alpha \in (T \cup N)^*$. First(α) est l'ensemble de tous les premiers terminaux des mots dérivés de α, i.e. $\{a \in T \mid \exists\ w,\ \alpha \to^\star aw\}$*

Définition 12.1.3 (Follow). *Soit $X \in N$. Follow(X) est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation, i.e. $\{a \in T \mid \exists\ u,\ w\ S \to^\star uXaw\}$.*

Théorème 12.1.1 (Tarski). *Soit A un ensemble fini muni d'une relation d'ordre $\leq$ et d'un plus petit élément ε. Toute fonction $f : A \to A$ croissante admet un plus petit point fixe.*

Démonstration. Comme ε est le plus petit élément, $\varepsilon \leq f(\varepsilon)$. Par croissance, $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$ pour tout k. Mais A étant fini, il existe un plus petit k_0 tel que $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$. En le notant a_0, on a bien un point fixe de f. Si b est un autre point fixe, par croissance, puisque $\varepsilon \leq b$, on a bien le résultat. ■

12.1.1 Principe du calcul de Null(X)

Proposition 12.1.1. *Pour calculer Null(α), il suffit de déterminer Null(X) pour $X \in N$. On a Null(X) ssi :*

- *Il existe une production $X \to \varepsilon$*
- *Il existe une production $X \to Y_1 \dots Y_m$ où Null(Y_i) pour tout i.*

Il s'agit d'un ensemble d'équations mutuellement récursives. Autrement dit, on cherche la plus petite solution d'une équation de la forme : $\vec{V} = F(\vec{V})$.

Démonstration.
- $\Rightarrow$ Par récurrence sur le nombre d'étapes du calcul de point fixe, on montre que si Null(X) alors $X \to^\star \varepsilon$
- $\Leftarrow$ Par récurrence sur le nombre d'étapes de la dérivation, on montre la réciproque.

■

Ici, on a $A = \{0,1\}^n$. On munit $\{0,1\}$ de l'ordre $0 \leq 1$ et A de l'ordre point à point. On a $\varepsilon = (0, \dots, 0)$. La fonction calculant Null(X) à partir des Null(X_i) est croissante, et le théorème de Tarski s'applique. On construit donc un point fixe à partir de ε.

12.1.2 Principe du calcul de First(X)

De même que pour Null : Les équations définissant First sont mutuellement récursives :

$$\textsc{First}(X) = \bigcup_{X \to \beta} \textsc{First}(\beta)$$

et :

$$\begin{aligned}
\textsc{First}(\varepsilon) &= \varnothing \\
\textsc{First}(a\beta) &= \{a\} \\
\textsc{First}(X\beta) &= \textsc{First}(X) \text{ si } \neg\textsc{Null}(X) \\
\textsc{First}(X\beta) &= \textsc{First}(X) \cup \textsc{First}(\beta) \text{ si } \textsc{Null}(X)
\end{aligned}$$

On applique alors le calcul de point fixe sur $A = \mathcal{P}(T)^n$ muni de $\subseteq$ point à point avec $\varepsilon = (\varnothing, \dots, \varnothing)$

12.1.3 Principe du calcul de Follow(X)

Les équations sont :

$$\textsc{Follow}(X) = \left(\bigcup_{Y \to \alpha X \beta} \textsc{First}(\beta) \right) \cup \left(\bigcup_{Y \to \alpha X \beta, \textsc{Null}(\beta)} \textsc{Follow}(Y) \right)$$

On procède par calcul de point fixe sur le même domaine que pour First.

12.2 Automate LR

12.2.1 $LR(0)$

Fixons pour l'instant $k = 0$. On commence par contruire un automate asynchrone :

— Les états sont des items de la forme $[X \to \alpha \cdot \beta]$ où $X \to \alpha\beta$ est une production de la grammaire. L'intuition est : « je cherche à reconnaître X, j'ai déjà lu α et je dois encore lire β ».

— Les transitions sont étiquetées par $T \cup N$ et sont les suivantes :

$$[Y \to \alpha \cdot a\beta] \to^a [Y \to \alpha a \cdot \beta]$$
$$[Y \to \alpha \cdot X\beta] \to^X [Y \to \alpha X \cdot \beta]$$
$$[Y \to \alpha \cdot X\beta] \to^\varepsilon [X \to \cdot\gamma] \text{ pour toute production } X \to \gamma$$

On déterminise ensuite l'automate en regroupant les états reliés par des ε-transitions. Les états de l'automate déterministe sont donc des ensembles d'items.
Par construction : chaque état s est saturé par la propriété : si $Y \to \alpha \cdot X\beta \in s$ et si $X \to \gamma$ est une production, alors $X \to \cdot\gamma \in s$. L'état initial est celui contenant $S \to \cdot E\#$.

On construit alors la table action :

— action$(s, \#)$ = succès si $[S \to E \cdot \#] \in s$

— action(s, a) = shifts' si $s \to^a s'$

— action(s, a) = reduce$X \to \beta$ si $[X \to \beta\cdot] \in s$ pour tout a.

— échec dans tous les autres cas

On construit alors la table goto : goto$(s, X) = s'$ si $s \to^X s'$.
La table $LR(0)$ peut contenir deux sortes de conflits : lecture/réduction et réduction/réduction.

Définition 12.2.1. *Une grammaire est dite $LR(0)$ so la table ainsi construite ne contient pas de conflit.*

La construction $LR(0)$ engendre très facilement des conflits. On va chercher à limiter les réductions : on pose action(s, a) = reduce$X\beta$ si et seulement si $[X \to \beta\cdot] \in s$ et $a \in \textsc{Follow}(X)$. On obtient la classe de grammaire $S(imple)LR(1)$.

12.2.2 $LR(1)$

Cette classe étant restrictive, on introduit une classe de grammaires encore plus large : $LR(1)$, au prix de tables encore plus grandes. Dans l'analyse $LR(1)$, les items ont la forme : $[X \to \alpha \cdot \beta, a]$
Les transitions de l'automate $LR(1)$ non déterministe sont :

$$[Y \to \alpha \cdot a\beta, b] \to^a [Y \to \alpha a \cdot \beta, b]$$
$$[Y \to \alpha \cdot a\beta, b] \to^a [Y \to \alpha a \cdot \beta, b]$$
$$[Y \to \alpha \cdot X\beta, b] \to^X [Y \to \alpha X \cdot \beta, b]$$
$$[Y \to \alpha \cdot X\beta, b] \to^\varepsilon [X \to \cdot\gamma, c] \text{ pour tout } c \in \textsc{First}(\beta b)$$

L'état initial est celui qui contient $[S \to \cdot\alpha, \#]$. On peut déterminiser l'automate et construire la table correspondante : on introduit une action de réduction pour (s, a) seulement lorsque s contient un item de la forme $[X \to \alpha\cdot, a]$.

Pour des questions de puissances de calcul, on introduit la classe $LALR(1)$, lookahead LR, qui est une approximation beaucoup utilisée dans les outils de la famille yacc.

Cinquième partie

Cours 5 : 27/10

13 Localisations

L'outil ocamllex maintient, dans la structure de type Lexing.lexbuf, la position courante dans le texte source qui est analysé. On peut alors obtenir la localisation de la dernère chaîne reconnue par ocamllex. On peut utiliser ces informations pour localiser les erreurs de syntaxe, mais aussi de potentielles erreurs lexicales comme une chaîne ou un commentaire non fermé.
L'outil Menhir [2] récupère ces informations et les fournit dans deux valeurs $startpos et $endpos, qui, dans une action sémantique, correspondent au début et à la fin du texte reconnu par la règle de grammaire. On peut alors stocker cette information dans l'arbre de syntaxe abstraite.

14 Analyse Syntaxique Elementaire

On va ici construire un analyseur syntaxique pour des expressions arithmétiques incluant :

— des constantes
— des additions
— des multiplications
— des parenthèses

On part d'un analyseur lexical, par exemple écrit avec ocamllex :

```
type token =
    | CONST of int
    | PLUS
    | TIMES
    | LEFTPAR
    | RIGHTPAR
    | EOF
```

et on veut obtenir un arbre de syntaxe abstraite :

```
type expr =
    | Const of int
    | Add of expr * expr
    | Mul of expr * expr
```

On écrit un parser dans le fichier Menhir, puis, juste en dessous dans le fichier, l'analyseur syntaxique.

Remarque 14.0.0.1. *Conseil : Commencer par écrire un pretty-printer, ici, en OcamL avec la bibliothèque Format :*

```
open Format
let rec print_expr fmt = function
  | Add (e1, e2) -> fprintf fmt "%a +@ %a" print_expr e1 print_expr e2
  | e            -> print_term fmt e
and print_term fmt = function
  | Mul (e1, e2) -> fprintf fmt "%a *@ %a" print_term e1 print_term e2
  | e            -> print_factor fmt e
and print_factor fmt = function
  | Const n -> fprintf fmt "%d" n
  | e       -> fprintf fmt "(@[%a@])" print_expr e
```

2. L'outil Cairn permet de visualiser l'analyse de Menhir.

L'analyseur syntaxique suit la même structure que la fonction d'affichage [3]. C'est ça, le bon conseil de Gilles Kahn :

```
let rec parse_expr () =
    let e = parse_term () in
    if !t = PLUS then (next (); Add (e, parse_expr ())) else e
and parse_term () =
    let e = parse_factor () in
    if !t = TIMES then (next (); Mul (e, parse_term ())) else e
and parse_factor () = match !t with
    | CONST n -> next (); Const n
    | LEFTPAR -> next ();
        let e = parse_expr () in
        if !t <> RIGHTPAR then error ();
        next (); e
    | _ -> error ()
```

Remarque 14.0.0.2. *On pourrait inclure l'analyse lexicale dans un tel code, avec d'autres fonctions récursives pour lire les constantes entières, ignorer les blancs, etc...*
Pour des opérateurs associatifs à gauche, le code sera légèrement différent mais le principe reste le même.

15 Analyse Descendante

15.1 Fonctionnement

On va procéder par expansions successives du non-terminal le plus à gauche (dérivation gauche) en partant de S et en utilisant une table qui indique pour un non-terminal X à expanser et les k premiers caractères de l'entrée, l'expansion $X \to \beta$ à effectuer. Dans la suite, on va prendre $k = 1$, et on va noter $T(X, c)$ cette table. En pratique, on suppose qu'un symbole terminal # dénote la fin de l'entrée, et la table indique donc également l'expansion de X lorsqu'on atteint la fin de l'entrée. On utilise une pile qui est un mot de $(N \cup T)^\star$. Initialement, la pile est réduite au symbole de départ. A chaque instant, on va ensuite examiner le sommet de la pile et le premier caractère c de l'entrée :

— Si la pile est vide, on s'arrête ; il y a succès ssi c est #.

— Si le sommet de la pile est un terminal a, alors a doit être égal à c, on dépile alors a et on consomme c ; sinon on échoue.

— Si le sommet de la pile est un non terminal X, on remplace X par le mot $\beta = T(X, c)$ en sommet de pile, en empilant les caractères de β en partant du dernier ; sinon on échoue

15.2 Programmation

Un analyseur descendant se programme en introduisant une fonction pour chaque non terminal de la grammaire. Chaque fonction examine l'entrée, et selon le cas, la consomme ou appelle récursivement les fonctions correspondant à d'autres non terminaux, selon la table d'expansion.

Remarque 15.2.0.1.
— *La table d'expansion n'est alors pas explicite : elle est dans le code de chaque fonction.*
— *La pile n'est pas explicite, elle est réalisée par la pile d'appels.*
— *En pratique il faut construire l'arbre de syntaxe abstraite.*

3. Le code n'est pas complet, cf page du cours

15.3 Construction de la Table d'Expansion

Pour décider si on réalise l'expansion $X \to \beta$ lorsque le premier caractère de l'entrée est c, on va chercher à déterminer si c fait partie des premiers caractères des mots reconnus par β. Une difficulté se pose pour une production telle que $X \to \varepsilon$, et il faut alors considérer l'ensemble des caractères qui peuvent suivre X. On utilise donc à nouveau les fonctions first et follow :Pour chaque production $X \to \beta$

— On pose $T(X, a) = \beta$ pour tout $a \in \text{first}(\beta)$

— Si null(β), on pose aussi $T(X, a) = \beta$ pour tout $a \in \text{follow}(X)$.

Il peut y avoir plusieurs éléments dans une même case :

Définition 15.3.1. *Une grammaire est dite $LL(1)$ si, dans la table précédente, il y a au plus une production dans chaque case.*

Une grammaire récursive gauche, i.e. contenant des productions de la forme : $X \to X\alpha|\beta$, ne sera jamais $LL(1)$. En effet, les first seront les mêmes pour ces deux productions. En particulier, la grammaire : $\begin{array}{rl} E \to & E + T \\ \to & T \\ T \to & T * F \\ \to & F \\ F \to & (E) \\ \to & \text{int} \end{array}$ n'est pas $LL(1)$. Plus généralement, si une grammaire contient $X \to a\alpha + a\beta$. Il faut alors factoriser (à gauche) les productions qui commencent par le même terminal.

16 Indentation comme Syntaxe

Dans certains langages, l'indentation (blancs de début de ligne/alignement vertical) est utilisée pour définir la syntaxe. L'analyseur lexical introduit des lexèmes NEWLINE (fin de ligne), INDENT (quand l'indentation augmente) et DEDENT (quand elle diminue). Il suffit alors de les utiliser dans la grammaire du langage.