Langages de Programmation, Interprétation, Compilation

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

Introduction

Organisation du cours

- Site ecampus du cours

  https://ecampus.paris-saclay.fr/course/view.php?id=185060
- Site ecampus du secrétariat (Sandrine Delahaye)

  https://ecampus.paris-saclay.fr/course/view.php?id=182108
- Cours conçu par Thibaut Balabonski

  https://public.lmf.cnrs.fr/~blsk/CompilationLDD3/
- Chargés de TD/TP : Jérémy Marrez, Paul Patault et Christine Paulin

Objectifs

Ce cours explore trois questions à propos des langages de programmation :

- leur définition par une sémantique et une, ou plutôt deux, syntaxe(s),
- leur interprétation, c'est-à-dire l'écriture d'une fonction qui prend en entrée un programme (donné par son code source) et ses éventuels arguments et renvoie le résultat de l'exécution du programme,
- leur compilation (traduction), c'est-à-dire l'écriture d'une fonction qui prend en entrée un programme (donné par son code source) et produit un programme équivalent écrit dans un autre langage.

Positionnement

- PIL (L2),
  - Commun : expressions régulières, automates, grammaires, analyse descendante, outils d'analyse lexicale et syntaxique, règles sémantiques
  - Nouveau : analyse ascendante, interprétation, langage assembleur
  - Différent : mise en œuvre Caml
- Langages formels (LF) L3 second semestre
  - Approfondissement des aspects théoriques de l'analyse ascendante
- IPF/PFA : connaissances et compétences en programmation fonctionnelle
- Logique : syntaxe/sémantique, termes, récurrence structurelle

Modalités de contrôle des connaissances

- Partiel 25%
- Projet 25%
- Examen final 50%

En session 2 : projet (reprise) 25% et examen 75%

Mise en garde

Emprunter sans citer c'est frauder !

- Conseil de discipline, relevé des notes suspendu à la décision
- Lettres de recommandation (?)

Ne cédez pas à la facilité !

- Savoir faire est important
- Connaître et comprendre l'est encore plus
- Les études (universitaires) sont faites pour cela
  (pas juste pour le diplôme et les notes), profitez-en !

Infos pratiques

- Cette semaine (1er septembre) : 2 cours
- Semaine prochaine (8 septembre) : 2 TP/TD (attention début à 8h30)
- Semaine standard :
  - en autonomie (avant dimanche midi) :
    - lecture d'un chapitre,
    - réponse quizz, questions
  - cours (explications, approfondissement. . .)
    - le lundi à 9h,
    - suivi d'un TD ou TP à 10h45

Planning prévisionnel

1/09 (*13h30*)	1. Panorama 2. Interprétation	
8/09 (*8h30*)		1. TP mise en route Ocaml 2. TD Induction structurelle
15/09	3. Syntaxe	3. TD Grammaires
22/09	4. Assembleur	4. TP assembleur MIPS
29/09	5. Analyse lexicale	5. TD expressions régulières
06/10	6. Analyse syntaxique	6. TD analyse ascendante
13/10	7. Outils d'analyse	7. TP lex/yacc (menhir)
20/10	*partiel*	
27/10	vacances	
03/11	8. Types	8. TD Types
10/11(**?**)	9. Fonctions	9. *TP projet*
17/11	10. Structures	10. TD assembleur MIPS (2)
24/11	11. Programmation Objet	11. *TP projet*
01/12	12. Compilation objet	12. TD révisions
08/12	réserve	
15/12	*examen*	

Bibliography

A.V. Aho, R. Sethi, and J. D. Ullman.
Compilers : principles, techniques and tools.
Addison Wesley, 1986.

Andrew W. Appel.
Modern Compiler Implementation in ML.
Cambridge University Press, 1998.

Benjamin C. Pierce.
Types and programming languages.
MIT Press, 2002.

Serres Michel (1930-2019).
Le trésor, dictionnaire des sciences.
Paris : Flammarion , DL 1997, 1997.

A propos de Ocaml

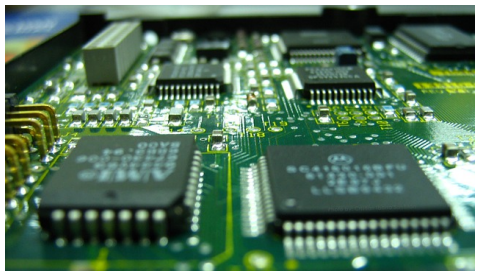
- Ocaml comme langage de description des concepts et algorithmes
- Pré-requis : cours IPF de L2 (voir intro/mise à niveau PFA mardi après-midi)
  - Introduction à la prog. fonctionnelle (IPF)
    `https://usr.lmf.cnrs.fr/~kn/ipf_fr.html`
  - Programmation fonctionnelle avancée (PFA)
    `https://usr.lmf.cnrs.fr/~kn/pfa_fr.html`
- Retour sur les principales notions en cours
- Premier TP de remise en jambes
- Références :
  - Site officiel (installation, manuels) `https://ocaml.org/`
  - Voir aussi le site IPF
  - Livre “Apprendre à programmer avec Ocaml”
    S. Conchon et J.-C. Filliâtre.
    Version anglaise “Learn Programming with OCaml” libre
    `https://usr.lmf.cnrs.fr/lpo/lpo.pdf`

Langages de Programmation

S'il vous plait, dessine-moi un mouton!

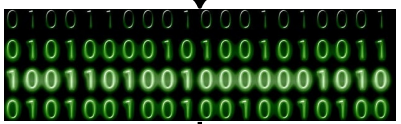
modélisation algorithmique

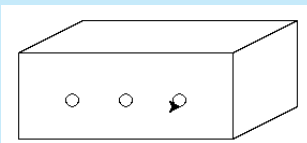
données
calcul



```
1   from turtle import *
2   def trou (x,y,r):
3       up()
4       goto(x,y)
5       down()
6       circle(r)
7   def mouton (h,L,l,a,t) :
8       goto(0,h)
9       down()
10      goto(0,0)
11      goto(L,0)
12      goto(L,h)
13      goto(0,h)
14      goto(l,h+a)
15      goto(L+l,h+a)
16      goto(L,h)
17      goto(L,0)
18      goto(L+l,a)
19      goto(L+l,h+a)
20      trou(L//4,t,6)
21      trou(2*L//4,t,6)
22      trou(3*L//4,t,6)
23      done()
24  mouton(80,200,60,30,30)
```

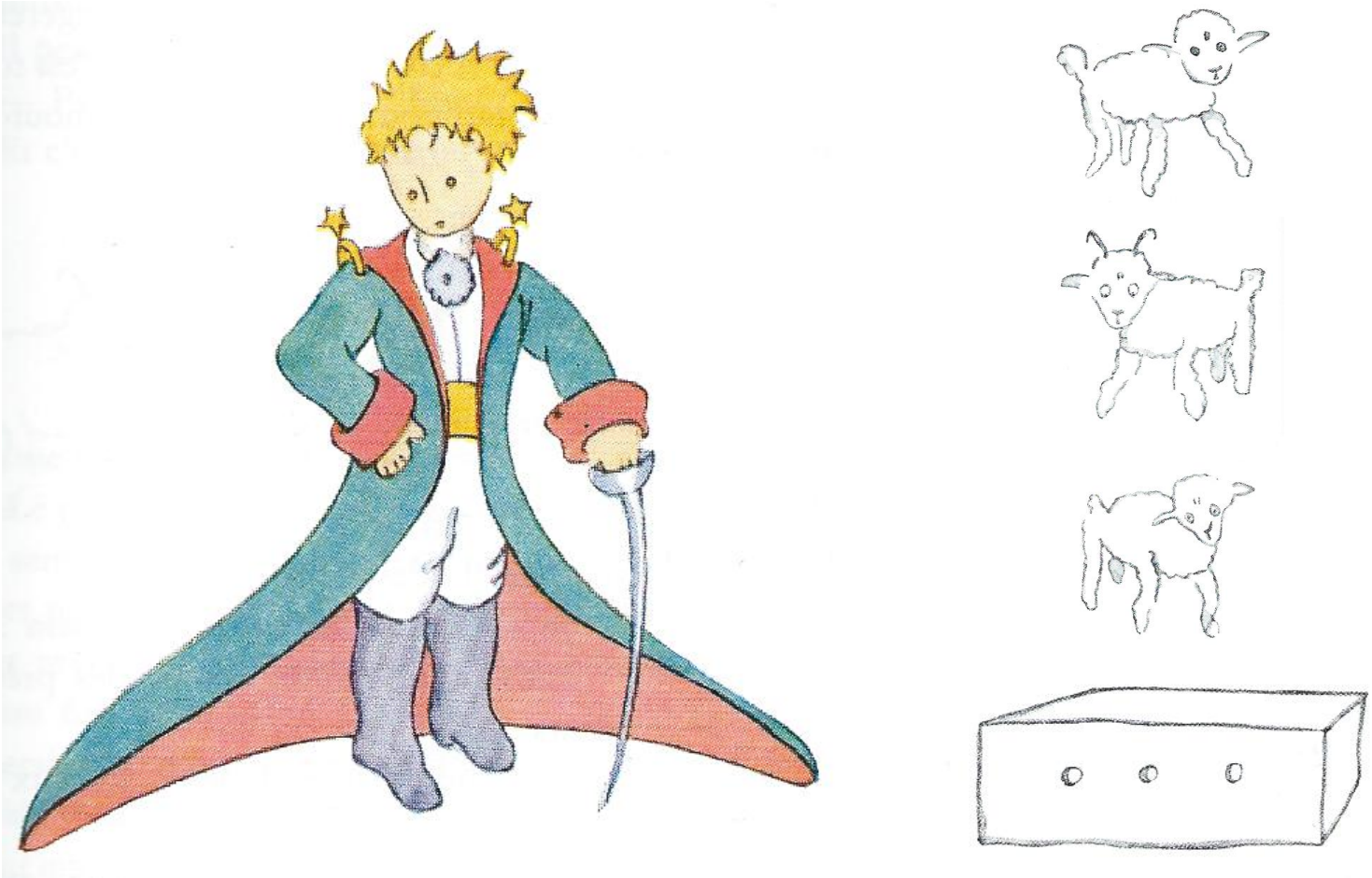
```
18          108 LOAD_GLOBAL              0 (goto)
            110 LOAD_FAST                1 (L)
            112 LOAD_FAST                2 (l)
            114 BINARY_ADD
            116 LOAD_FAST                3 (a)
            118 CALL_FUNCTION            2
            120 POP_TOP

19          122 LOAD_GLOBAL              0 (goto)
            124 LOAD_FAST                1 (L)
            126 LOAD_FAST                2 (l)
            128 BINARY_ADD
            130 LOAD_FAST                0 (h)
            132 LOAD_FAST                3 (a)
            134 BINARY_ADD
            136 CALL_FUNCTION            2
            138 POP_TOP

20          140 LOAD_GLOBAL              2 (trou)
            142 LOAD_FAST                1 (L)
            144 LOAD_CONST               2 (4)
            146 BINARY_FLOOR_DIVIDE
            148 LOAD_FAST                4 (t)
            150 LOAD_CONST               3 (6)
            152 CALL_FUNCTION            3
            154 POP_TOP

21          156 LOAD_GLOBAL              2 (trou)
            158 LOAD_CONST               4 (2)
            160 LOAD_FAST                1 (L)
            162 BINARY_MULTIPLY
            164 LOAD_CONST               2 (4)
            166 BINARY_FLOOR_DIVIDE
            168 LOAD_FAST                4 (t)
            170 LOAD_CONST               3 (6)
            172 CALL_FUNCTION            3
            174 POP_TOP
```





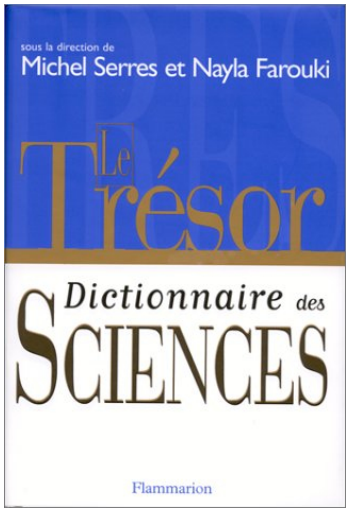
Référence

Le petit prince d'Antoine de Saint Exupery



Langages, Programmes

Programme
Programmation
Langage



Un programme est un texte, l'histoire qu'il raconte un ALGORITHME, la programmation son écriture. D'une rédaction à l'autre, le langage varie, il en est plusieurs milliers, le style aussi : il y a différentes manières de rédiger un même algorithme. L'étude de la programmation est celle de ces styles : ils ont préludé à la prolifération des langages de programmation, ils en sont le prétexte et la justification. Avant ces langages, il n'y avait que la contrainte : le langage machine n'est que Le reflet de la structure des ordinateurs : le langage d'assemblage en facilite l'accès, sans toutefois dégager un style propre qui tranche, car il lui manque la concision.

. . .

Par la suite, maints langages de programmation et, pour chacun, maints dialectes se spécialisèrent dans différents usages et proposèrent de nouvelles manières, Les usages sont foison : ils ne relèvent pas de la programmation à proprement parler. En revanche, les styles d'écriture en sont le cœur ; c'est d'eux que s'occupe la programmation, afin que les programmes informatiques soient aussi transparents aux machines qui les exécuteront qu'aux hommes, qui seront alors en mesure de les relire sans effort, de les modifier, voire de les corriger.

A propos des Langages et des Programmes (suite)

A la différence des programmes politiques, rarement réalisés tels qu'ils étaient prévus avant les élections, à la différence des programmes scolaires, toujours trop chargés et que l'on ne termine jamais, les programmes informatiques sont exécutés scrupuleusement jusqu'à leur terme, s'il en est un, et indéfiniment sinon. C'est ce qui fait la grandeur et la faiblesse des informaticiens. Grandeur, parce que leurs programmes sont indéfectiblement suivis d'effet ; faiblesse, parce que ces effets ne dépendent que d'eux, où tout au moins, de leurs programmes, ce qui leur renvoie, en permanence, en miroir, une image navrante de leurs propres insuffisances.

. . .

Langues et langages sont faits pour exprimer, cest-à-dire pour manifester quelque chose ; on doit donc dissocier, dans toute langue et dans tout langage, les expressions, qui sont des signes, de ce qu'elles désignent qui constitue leur sens. L'étude des expressions relève de la grammaire qui recense l'ensemble des formulations possibles dans une langue où dans un langage, à savoir l'ensemble de ce qui y a possiblement sens. La détermination du sens des expressions incombe, quant à elle, à la SEMANTIQUE.

Notons que la grammaire distingue le lexique, qui inventorie les briques élémentaires à partir desquelles se construisent les expressions, et la syntaxe, qui aborde l'agencement de ces briques.

Partie 1 : Introduction

Panorama

Description d'un calcul (calculatrice, entrée avancée...)

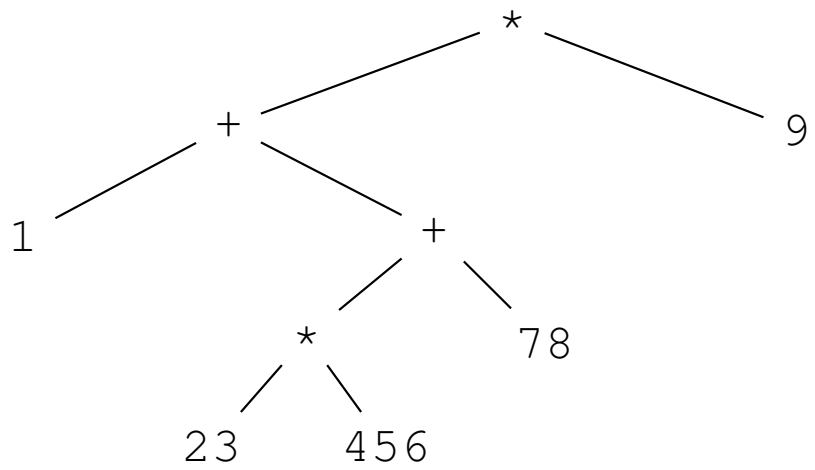
- Entrée chaîne de *caractères*

  $(1 + 23 * 456 + 78) * 9 \quad \rightsquigarrow \quad (, 1, +, 2, 3, *, 4, 5, 6, +, 7, 8,), *, 9$

- Analyse lexicale : suite d'*éléments* pertinents :

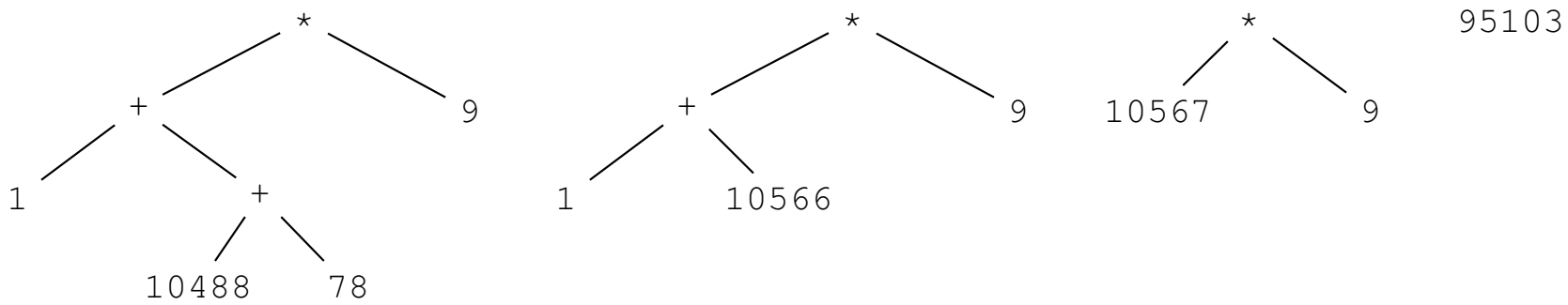
  $(, 1, +, 23, *, 456, +, 78,), *, 9$

- Analyse grammaticale (syntaxique) : arbre



Calcul

- Transformation de l'arbre (des feuilles vers la racine)



- Compilation en assembleur

```
li   $t0, 1              # t0 <- 1
li   $t1, 23             # t1 <- 23
li   $t2, 456            # t2 <- 456
mul  $t1, $t1, $t2       # t1 <- t1 * t2
li   $t2, 78             # t2 <- 78
add  $t1, $t1, $t2       # t1 <- t1 + t2
add  $t0, $t0, $t1       # t0 <- t0 + t1
li   $t1, 9              # t1 <- 9
mul  $t0, $t0, $t1       # t0 <- t0 * t1
```

Analyse lexicale

- De la suite de caractères vers une suite de *mots* représentant des entités (unités) lexicales (token, léxème)
- Il y a un nombre fini de classes différentes
- A une classe peut être associée une valeur (cas des constantes)
- Code Ocaml

```
type token =
    | NUM of int
    | PLUS
    | STAR
    | LPAR
    | RPAR
let tokenize (s: string): token list = ...
```

Arbre de syntaxe abstraite

- Type

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

- Evaluation

```
let rec (eval : expr -> int) = function
  | Cst n        -> n
  | Add(e1, e2) -> eval e1 + eval e2
  | Mul(e1, e2) -> eval e1 * eval e2
```

Analyse grammaticale

- Reconnaître les entrées correctes, rejeter les autres
- Gérer les ambiguïtés suivant les règles de priorités
- Produire l'arbre de syntaxe abstraite

Algorithme de la gare de triage (shunting yard, Dijkstra, 1961)

- entrée (suite de léxèmes), pile d'expressions reconnues, pile d'opérateurs en attente
- lecture de l'entrée de gauche à droite
- fin : entrée vide, plus d'opérateurs et une seule expression

Shunting yard : fonctionnement

- nombre : transféré sur la pile des expressions
- opérateur : placé en attente sur la pile d'opérateurs jusqu'à ce que l'opérande droit soit analysé
- deux expressions et un opérateur au sommet de leurs piles respectives sont regroupés pour former une seule expression, sauf si le prochain opérateur de l'entrée est plus prioritaire que l'opérateur vu sur la pile d'opérateurs.
- parenthèse ouvrante : placée sur la pile des opérateurs en attendant la parenthèse fermante associée

Shunting yard : exemple

entree	ops	exprs
2 * 3 + 4 * 5	.	.
* 3 + 4 * 5	.	2
3 + 4 * 5	*	2
+ 4 * 5	*	3 2
+ 4 * 5	.	(2*3)
4 * 5	+	(2*3)
* 5	+	4 (2*3)
5	* +	4 (2*3)
.	* +	5 4 (2*3)
.	+	(4*5) (2*3)
.	.	((2*3)+(4*5))

Version structurée

- Caractérisation
  - une expression générale est une somme de une ou plusieurs expressions multiplicatives,
  - une expression multiplicative est un produit de une ou plusieurs expressions atomiques,
  - une expression atomique est une constante entière, ou une expression générale placée entre parenthèses.
- Code

```
parse_expr, parse_m_expr, parse_atom
          : token list -> expr * token list
```

- Fonctions auxilliaires pour gérer les suites d'additions et de multiplications

```
expand_sum, expand_prod
          : expr * token list -> expr * token list
```

Compilation

- Machine virtuelle

```
type instr =
  | ICst of int   (* place constante sur la pile *)
  | IAdd  (* additionne deux valeurs en tête de pile *)
  | IMul  (* multiplie deux valeurs en tête de pile *)
```

- Evaluation par une machine à pile

```
let exec (code : instr list -> int) =
  let rec (ex : instr list * int list -> int = function
    | ([], v :: stack) -> assert (stack = []); v
    | (ICst n :: code, stack) ->
        ex(code,n :: stack)
    | (IAdd :: code, v2 :: v1 :: stack) ->
        ex(code,v1+v2::stack)
    | (IMul :: code, v2 :: v1 :: stack) ->
        ex(code, v1*v2 :: stack)
    | _ -> failwith "empty stack"
    in  ex(code, [])
```

Génération de code

```
let rec codegen: expr -> instr list = function
  | Cst n          -> [ICst n]
  | Add(e1, e2) -> codegen e1 @ codegen e2 @ [IAdd]
  | Mul(e1, e2) -> codegen e1 @ codegen e2 @ [IMul]
```

Utiliser des traits impératifs

- Fonction qui transforme une entrée en sa sortie
- Procédure qui modifie en place une référence
- Spécification
  - Fonction f : relation entre x et $f(x)$
  - Procédure avec effet sur r : relation entre la valeur de r avant la procédure et après la procédure
- Utilisation de références locales (en particulier pour les fonctions auxiliaires)
  - écriture allégée
  - exemples : indice de position dans l'entrée, code généré, pile pour les calculs. . .

Analyse lexicale incrémentale

- Au lieu de produire la suite des léxèmes, une fonction qui nous donne le prochain
- Ajout d'un léxème spécifique pour marquer la fin de l'entrée

Synthèse

- A retenir :
  - les différentes phases de l'analyse d'une expression
    - analyse lexicale
    - analyse grammaticale
    - interprétation/compilation
  - les données manipulées
    - chaîne de caractères
    - unités lexicales
    - arbre de syntaxe abstraite
    - instructions machine
- Savoir faire :
  - Lire et comprendre des déclarations de types et fonctions en Ocaml
  - Déterminer le type d'un programme (simple)
  - Savoir adapter types et fonctions à des constructions nouvelles
  - Usage du type des listes, référence simple

Partie 1 : Introduction

1. A propos d'une calculatrice
2. Interprétation d'un langage impératif
   - Syntaxe concrète et syntaxe abstraite
   - Structures inductives
   - Variables et environnements
   - Variables mutables et instructions
   - Approfondissement : interprète complet pour IMP
   - Approfondissement : subtilités sémantiques

Introduction : langage IMP

Langage impératif minimal : IMP

```
a := 2;
n := 6;
r := 1;
while (0 < n) {
  if (n % 2 == 1) {
    r := r*a;
  } else {}
  a := a*a;
  n := n/2;
}
print(r);
```

- Distinction entre expression et instruction
- Sémantique : usuelle
- L'instruction `print` affiche un caractère donné par son code ASCII.

Syntaxe concrète versus syntaxe abstraite

- syntaxe concrète : ce que le programmeur écrit : une chaîne de caractères
  partie visible du langage
- syntaxe abstraite : représentation structurée du programme sous forme d'arbre
  le cœur du langage, utile à la sémantique, la génération de code
  - les espaces, commentaires, parenthèses pour la désambiguation, caractères de structuration ne sont pas conservés
  - *Sucre syntaxique* : certaines facilités d'écriture sont directement traduites
    let f x = e définit une variable à laquelle on affecte une fonction anonyme :
    let f = **fun** x -> e

Objets inductifs

On définit un ensemble d'objets inductifs en décrivant :

1. des objets de base,
2. un ensemble fini de constructeurs, destinés à combiner des objets déjà construits pour en définir de nouveaux.

On s'intéresse alors à l'ensemble des objets qui peuvent être construits en utilisant les deux points précédents.

- On appelle arité d'un constructeur le nombre d'éléments qu'il combine.
- Les objets de base peuvent être vus comme des constructeurs d'arité zéro.
- L'ensemble des objets de base et des constructeurs définissant un tel ensemble est appelé sa signature.

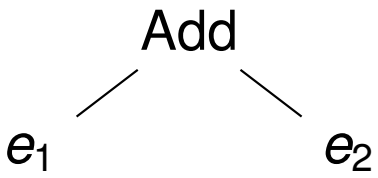
Exemple : expressions arithmétiques.

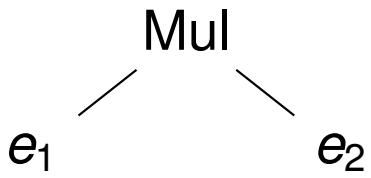
Ensemble minimal $\mathbb{A}$d'expressions arithmétiques avec :

- les constantes entières comme objets de base,

$$0 \qquad 1 \qquad 2 \qquad 3 \qquad \cdots$$

- quelques constructeurs binaires, c'est-à-dire combinant chacun deux expressions déjà construites, par exemple pour l'addition et la multiplication.





Notation concrète pour syntaxe abstraite

- La syntaxe abstraite est une structure arborescente
- Notation concrète nécessaire pour les manipulations
  Alléger la manipulation à l'aide de notations mathématiques comme n, $e_1 \oplus e_2$ et $e_1 \otimes e_2$.
- On utilise $\oplus$ et $\otimes$ pour les constructions syntaxiques
  distincts des opérateurs $+$ et $\times$ classiques, pour l'interprétation.
  - L'opérateur $+$ est un élément mathématique, qui s'applique à deux nombres pour en calculer un troisième. Il vérifie l'équation $1 + 2 = 3$.
  - Le constructeur $\oplus$ est un élément syntaxique, qui s'applique à deux expressions pour en construire une troisième. Il sert à définir le langage lui-même, dans lequel $1 \oplus 2 \neq 3$.
    Ajout systématiques de parenthèses pour lever les ambiguïtés

Exemple : listes

Les listes peuvent êtres vues comme un ensemble d'objets inductifs défini par :

- un unique objet de base : la liste vide, notée [],
- un unique constructeur, qui crée une nouvelle liste $e :: \ell$ en ajoutant un élément e en tête d'une liste ℓ.

Le constructeur attend deux arguments mais le premier est un paramètre dans un ensemble quelconque, ce n'est pas une liste.
La définition se comporte comme si on avait une famille de constructeurs de listes unaires, un pour chaque valeur de l'élément ajouté en tête.

Définitions récursives

Dans un ensemble E d'objets inductifs, chaque objet peut être construit en utilisant uniquement les objets de base et les constructeurs. Pour définir une fonction f s'appliquant aux éléments de E, il suffit donc :

- pour chaque élément de base e, donner $f(e)$,
- pour chaque constructeur n-aire c, exprimer $f(c(e_1, \ldots, e_n))$ en fonction des sous-éléments e_i et, de leurs images $f(e_i)$ par la fonction, pour chaque $e_i \in E$.

Ainsi, on aura défini une image unique pour chaque élément de E.

Définitions récursives : exemples

$$\begin{cases} \text{nbCst}(n) & = & 1 \\ \text{nbCst}(e_1 \oplus e_2) & = & \text{nbCst}(e_1) + \text{nbCst}(e_2) \\ \text{nbCst}(e_1 \otimes e_2) & = & \text{nbCst}(e_1) + \text{nbCst}(e_2) \end{cases}$$

$$\begin{cases} \text{nbOp}(n) & = & 0 \\ \text{nbOp}(e_1 \oplus e_2) & = & 1 + \text{nbOp}(e_1) + \text{nbOp}(e_2) \\ \text{nbOp}(e_1 \otimes e_2) & = & 1 + \text{nbOp}(e_1) + \text{nbOp}(e_2) \end{cases}$$

$$\begin{cases} \text{eval}(n) & = & n \\ \text{eval}(e_1 \oplus e_2) & = & \text{eval}(e_1) + \text{eval}(e_2) \\ \text{eval}(e_1 \otimes e_2) & = & \text{eval}(e_1) \times \text{eval}(e_2) \end{cases}$$

Programmer avec des objets inductifs

- Traduction directe des ensembles inductifs en types “algébriques”
- Définition récursive en utilisant la récursivité et le filtrage
- Ocaml permet une récursion et un filtrage plus avancés que la définition mathématique
  - vérification à la compilation que le filtrage est complet
  - aucune vérification sur les appels récursifs
- Les types algébriques de Ocaml sont plus puissants que nos objets inductifs
  - objets infinis
  - programmes non récursifs qui ne terminent pas

Raisonnement par induction structurelle

Chaque objet d'un ensemble E d'objets inductifs peut être construit (de manière finie) en utilisant uniquement les objets de base et les constructeurs. Pour démontrer qu'une propriété P est vraie pour tous les éléments de E, il suffit donc :

- de démontrer que $P(e)$ est vraie pour chaque élément de base e,
- de démontrer que si on prend un constructeur n-aire c et n éléments e_1, ..., e_n quelconques, si on suppose de plus que $P(e_i)$ est vérifié pour tous les $e_i \in E$, alors la propriété P est encore vraie pour l'élément construit $c(e_1, \ldots, e_n)$.

Avec ces hypthèses, on assure qu'il est impossible de construire un élément e qui ne vérifierait pas la propriété.
Cette technique de preuve est une preuve par récurrence, qu'on appelle plus précisément preuve par récurrence structurelle, ou encore preuve par induction structurelle.
Dans le deuxième point, les hypothèses $P(e_{i_1})$ à $P(e_{i_k})$ que l'on peut utiliser pour justifier $P(c(e_1, \ldots, e_n))$ sont les hypothèses de récurrence, ou hypothèse d'induction.

Exemples

Listes : Une propriété P est vraie pour toutes les listes, si :

1. elle est vraie pour la liste vide [],
2. quels que soient la liste ℓ et l'élément e, si P est vraie pour ℓ (hypothèse de récurrence), alors elle est encore vraie pour $e :: \ell$.

Expressions arithmétiques : Une propriété P est vraie pour toutes les expressions arithmétiques, si :

1. elle est vraie pour toutes les constantes entières,
2. quelles que soient les expressions e_1 et e_2, si P est vraie pour e_1 et pour e_2 (hypothèses de récurrence), alors elle est encore vraie pour $e_1 \oplus e_2$,
3. quelles que soient les expressions e_1 et e_2, si P est vraie pour e_1 et pour e_2 (hypothèses de récurrence), alors elle est encore vraie pour $e_1 \otimes e_2$.

Exemple de preuve

Montrons que dans toute expression arithmétique, le nombre de constantes est de un supérieur au nombre d'opérateurs.
On note $P(e)$ la propriété $\text{nbCst}(e) = \text{nbOp}(e) + 1$
On vérifie les cas de base et récursifs pour chaque symbole de la signature :

- Base : pour tout terme constant n on a $\text{nbCst}(\text{n}) = 1$ et $\text{nbOp}(\text{n}) = 0$.
  La propriété P est bien vérifiée pour le terme n.
- Cas de l'addition (cas récursif) : soient deux expressions e_1 et e_2 pour lesquelles la propriété P est vraie. On a alors

$$
\begin{array}{lll}
\text{nbCst}(e_1 \oplus e_2) & & \\
= & \text{nbCst}(e_1) + \text{nbCst}(e_2) & \text{par définition de nbCst} \\
= & (\text{nbOp}(e_1) + 1) + (\text{nbOp}(e_2) + 1) & \text{par hypothèses de récurrence} \\
= & (1 + \text{nbOp}(e_1) + \text{nbOp}(e_2)) + 1 & \text{(réarrangement)} \\
= & \text{nbOp}(e_1 \oplus e_2) + 1 & \text{par définition de nbOp}
\end{array}
$$

  La propriété P est encore vraie pour le terme $\text{Add}(e_1, e_2) = e_1 \oplus e_2$.
- Cas de la multiplication (cas récursif) : similaire au cas de l'addition.

On a donc démontré par récurrence structurelle que pour toute expression arithmétique e on a bien $\text{nbCst}(e) = \text{nbOp}(e) + 1$.

Variables et environnements

Une variable est un nom désignant une valeur stockée en mémoire. En mathématiques, une variable est une indéterminée

```
int x := 3;
int y := 1 + 2 * x;
return 2 * x * y;
```

```
let x = 3 in
let y = 1 + 2 * x in
2 * x * y
```

Évaluation des expressions avec variables.

- Pour représenter les variables, il suffit d'ajouter à la signature des expressions un symbole d'arité 0 pour chaque nom de variable.
- La fonction eval d'évaluation des expressions a besoin d'une information sur les valeurs associées aux différentes variables en mémoire.
  - Abstraction : fonction ρ appelée environnement, qui à chaque nom de variables associe sa valeur.

  La fonction eval prend un environnement en deuxième paramètre, à laquelle elle fait appel pour obtenir la valeur des variables.

$$\begin{cases} \mathrm{eval}(\mathsf{n}, \rho) & = & n \\ \mathrm{eval}(\mathsf{x}, \rho) & = & \rho(\mathsf{x}) \\ \mathrm{eval}(e_1 \oplus e_2, \rho) & = & \mathrm{eval}(e_1, \rho) + \mathrm{eval}(e_2, \rho) \\ \mathrm{eval}(e_1 \otimes e_2, \rho) & = & \mathrm{eval}(e_1, \rho) \times \mathrm{eval}(e_2, \rho) \end{cases}$$

Version fonctionnelle : variables locales immuables

En caml, les variables sont immuables : elles reçoivent lors de leur déclaration une valeur, qui n'est jamais modifiée.
Evaluation de l'expression **let** y = 1+2*x **in** 2*x*y

- l'expression 1+2*x est évaluée dans un certain environnement ρ,
- puis l'expression 2*x*y est évaluée dans un nouvel environnement ρ', qui étend ρ en y ajoutant l'association d'une valeur à la variable y.
- La variable y est locale à l'expression 2*x*y située à droite du **in**, elle n'existe pas dans les autres parties du programme.
  L'environnement étendu ρ' n'est utilisé que pour l'évaluation de cette sous-expression.

Pour un environnement ρ, une variable x et une valeur v, notons $\rho[x \mapsto v]$ l'environnement ρ augmenté d'une association de x à v. Cet environnement ρ' vérifie donc $\rho'(x) = v$, et $\rho'(y) = \rho(y)$ pour tout $y \neq x$.

$$\text{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho) \quad = \quad \text{eval}(e_2, \rho[x \mapsto v]) \qquad \text{avec } v = \text{eval}(e_1, \rho)$$

Représentation en caml

On étend le type caml des expressions arithmétiques avec deux constructeurs pour l'accès à et la déclaration d'une variable locale.

```
type expr =
  ...
  | Var of string
  | Let of string * expr * expr
```

L'expression let x = 41 in x + 1 est représentée par la valeur caml

```
Let("x", Cst 41, Add(Var "x", Cst 1))
```

Valeurs et environnements

Type value pour représenter les valeurs produites par les expressions arithmétiques.

```
type value = int
```

Un environnement associe des noms de variables (c'est-à-dire des chaînes de caractères) à des valeurs (de type value). Structure de table associative, plusieurs implantations possibles :

- arbres de recherche équilibrés (module Map de caml), structure de données immuable, en style purement fonctionnel,
- les tables de hachage (module Hashtbl de caml), qui forment une structure de données mutable.

Solution à base d'arbres de recherche équilibrés

```
module Env = Map.Make(String)
type env = value Env.t
```

- Après cette déclaration, le type env désigne des tables associatives avec des clés de type string et des valeurs de type value.
- Le module Env fournit une constante Env.empty pour une table vide
- nombreuses fonctions, dont Env.find pour la récupération de la valeur associée à une clé dans une table, ou Env.add pour l'ajout ou le remplacement d'une association à une table.

Voir code de l'interprète

Variables mutables et instructions

Une variable est un nom désignant une valeur stockée en mémoire.

```
int x := 3;
int y := 1 + 2 * x;
return 2 * x * y;
```

```
let x = 3 in
let y = 1 + 2 * x in
2 * x * y
```

- Cas des variables immuables, qui reçoivent une valeur définitive à leur création.
- Dans un langage comme C, python, java, ou IMP les variables sont au contraire mutables : les instructions successives d'un programme peuvent modifier à volonté la valeur des variables de ce programme. Les instructions modifient l'environnement pour la suite du programme

Instructions

- En plus des expressions, on définit donc un nouvel ensemble d'objets inductifs représentant les instructions.
- construction binaire $\mathtt{x} := e$ désignant une instruction d'affectation.
- Une séquence d'instructions est soit une instruction seule i, soit une instruction i suivie d'une séquence s, notée $i \ ; \ s$.
- L'exécution des instructions peut être décrite par une nouvelle fonction exec, qui s'applique à une instruction ou une séquence d'instructions et à un environnement.
- Pour modéliser le fait que l'effet d'une instruction est de modifier l'environnement avant l'exécution des instructions suivantes, on va définir comme résultat de exec l'environnement ρ' modifié. Ainsi

$$\begin{cases} \mathrm{exec}(\mathtt{x} := e, \rho) & = & \rho[x \mapsto v] & \text{avec } v = \mathrm{eval}(e, \rho) \\ \mathrm{exec}(i \ ; \ s, \rho) & = & \mathrm{exec}(s, \rho') & \text{avec } \rho' = \mathrm{exec}(i, \rho) \end{cases}$$

Question

Quelle différence entre les comportements des programmes suivants en terme d'occupation mémoire (p représente une suite du programme quelconque)

```
x := 1;
x := 2;
p
```

```
let x = 1 in
let x = 2 in
p
```

```
let x' = 1 in
let x = 2 in
p
```

Exemple

Calculons l'état de l'environnement après l'exécution de la séquence d'affectations suivante, en prenant un environnement initialement vide.

```
x := 6;
y := x + 1;
x := x * y;
```

$$
\begin{aligned}
& \mathrm{exec}(\texttt{x := 6; y := } x \oplus 1\texttt{; x := } x \otimes y,\ \emptyset) \\
= \quad & \mathrm{exec}(\texttt{x := } x \otimes y,\ \mathrm{exec}(\texttt{y := } x \oplus 1,\ \mathrm{exec}(\texttt{x := 6},\ \emptyset))) \\
= \quad & \mathrm{exec}(\texttt{x := } x \otimes y,\ \mathrm{exec}(\texttt{y := } x \oplus 1,\ [x \mapsto 6])) \\
& \qquad \mid \mathrm{car}\ \mathrm{eval}(6, \emptyset) = 6 \\
= \quad & \mathrm{exec}(\texttt{x := } x \otimes y,\ [x \mapsto 6, y \mapsto 7]) \\
& \qquad \mid \mathrm{car}\ \mathrm{eval}(x \oplus 1, [x \mapsto 6]) = 7 \\
= \quad & [x \mapsto 42, y \mapsto 7] \\
& \qquad \mid \mathrm{car}\ \mathrm{eval}(x \otimes y, [x \mapsto 6, y \mapsto 7]) = 42
\end{aligned}
$$

Représentation en caml et interprétation

Voir code

Variante avec structure de données mutable

```
type value = int
type env = (string, value) Hashtbl.t
```

- La fonction d'évaluation est quasiment identique,
  - utilise cette fois la fonction find de la bibliothèque Hashtbl
- Les nouvelles fonctions exec_instr et exec_seq prennent toujours un environnement env en paramètre, mais n'ont plus besoin de renvoyer l'environnement modifié en résultat.
  - À la place, on effectue des modifications en place de env, ici à l'aide de la fonction de mise à jour Hashtbl.replace: env –> string –> value –> unit env

Interprète complet pour IMP

- Prise en compte des conditionnelles
  - Valeur d'une comparaison (booléen, entier $0/1$, entier quelconque ...
- Boucle while (interprétation qui peut ne pas terminer)

Subtilités sémantiques

- Opérateurs paresseux : les deux opérandes ne sont pas systématiquement évaluées (versus opérateurs stricts)
- Polysémie des variables et expressions mutables : adresse mémoire (valeur gauche) ou bien valeur stockée à l'adresse mémoire (valeur droite).
- Possibilité d'aliasing : deux noms de variables du programme désignent le même emplacement mémoire
  Quelle valeur de y après le programme :

  $$x := 1 \; ; \quad y := x \; ; \quad x := 2$$

  Il faut alors découper l'environnement en deux
  - association d'une adresse mémoire à un nom de variable
  - association d'une valeur à une adresse mémoire

Synthèse

- A retenir :
  - Définition d'ensemble inductif d'objets
    - caractérisation à l'aide des constructeurs
    - définition récursive de fonctions
    - preuve par récurrence structurelle
  - Rôle des environnements
  - Déroulement de l'exécution d'un programme
- Savoir faire :
  - Définition de fonctions sur des arbres de syntaxes abstraites
  - Implémenter une fonction d'évaluation à partir d'une caractérisation sémantique