Langages de programmation, interprétation, compilation

Thibaut Balabonski @ Faculté des Sciences d'Orsay. V3, automne 2023

Partie 1

Table des matières

1 À propos d'une calculatrice

Tour d'horizon où l'on réalise des fonctions d'interprétation et de compilation pour des expressions arithmétiques.

1.1 Panorama

Supposons que l'on s'intéresse à l'expression arithmétique

```
(1+23*456+78)*9
```

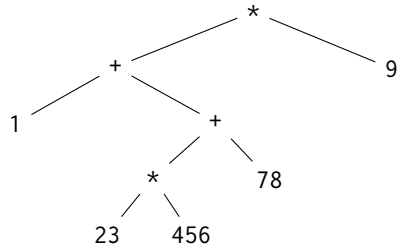
Une fois saisie sur le clavier d'une calculatrice ou d'un ordinateur, cette expression prend la forme d'une chaîne de caractères, formée de la suite de caractères suivante.

```
(, 1, +, 2, 3, *, 4, 5, 6, +, 7, 8, ), *, 9
```

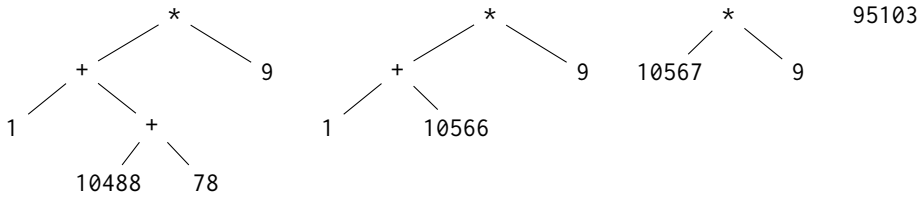
Cette séquence est plate, et ses éléments atomisés. Nous allons devoir l'analyser pour reconstruire sa structure et la manipuler réellement comme une expression arithmétique et non comme une simple suite de symboles (***analyse syntaxique***). La première étape, appelée ***analyse lexicale***, consiste à regrouper les symboles formant ensemble un même élément. On obtient par exemple le découpage suivant.

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

La deuxième étape, appelée ***analyse grammaticale***, consiste à associer correctement les différentes parties de l'expression aux bons opérateurs. Le résultat peut être présenté sous la forme d'un arbre.



Si l'on souhaite ***interpréter*** cette expression, c'est-à-dire calculer son résultat, on peut alors effectuer les différentes opérations en partant des feuilles de l'arbre et en remontant vers la racine.



Alternativement, on peut ***compiler*** cette expression, c'est-à-dire créer une séquence d'instructions élémentaires qu'une machine devra exécuter pour calculer le résultat. À nouveau, on peut faire cette traduction en se laissant guider par la structure de l'arbre. Voici une traduction possible dans un langage assembleur (code à gauche, et descriptif des effets obtenus à droite).

```
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
```

Un tel code assembleur peut ensuite être directement transformé en un fichier binaire exécutable par une machine (ici, de type MIPS).

1.2 Lexèmes et analyse lexicale

Faisons la liste des éléments pouvant apparaître dans l'écriture d'une expression arithmétique :

— des nombres (on prendra des entiers positifs),
— des opérateurs (on prendra + et *),
— des parenthèses ouvrantes ou fermantes.

On peut définir un type Caml permettant de décrire de tels éléments, qu'on appellera ici des ***mots*** (en jargon des ***lexèmes***, ou ***token*** en anglais).

```
type token =
  | NUM of int
  | PLUS
  | STAR
  | LPAR
  | RPAR
```

L'analyse lexicale peut donc être vue comme une fonction prenant une chaîne de caractères et renvoyant une liste de mots.

```
let tokenize (s: string): token list =
```

On itère sur les caractères de la chaîne avec une fonction récursive `lex` prenant en paramètre l'indice `i` du caractère courant. À noter : la définition de la fonction `lex` étant à l'intérieur de la fonction `tokenize`, on a directement accès à la chaîne `s` donnée en paramètre à `tokenize`.

```
  let rec lex i =
    if i >= String.length s then
      []
```

Si l'indice `i` n'a pas dépassé le dernier caractère, on observe ce caractère courant. Dans le cas d'un caractère désignant à lui seul un mot, on combine ce mot et la liste obtenue en poursuivant la boucle à partir du caractère suivant.

```
    else match s.[i] with
      | '+' -> PLUS :: (lex (i+1))
      | '*' -> STAR :: (lex (i+1))
      | '(' -> LPAR :: (lex (i+1))
      | ')' -> RPAR :: (lex (i+1))
```

Une espace est tout simplement ignorée : la boucle continue au caractère suivant et rien n'est ajouté à la liste des mots reconnus.

```
      | ' ' -> lex (i+1)
```

Si le caractère courant est un chiffre, on va devoir lire une séquence de chiffres pour former un nombre. Pour cela, on fait appel à une fonction auxiliaire `num` chargée de lire l'entrée jusqu'à la fin du nombre et qui renvoie, d'une part le nombre `n` lu, et d'autre part l'indice `i` du caractère situé immédiatement après. On poursuit ensuite l'analyse à partir de cet indice.

```
      | '0'..'9' -> let n, i = num 0 i in
                    NUM n :: lex i
```

Enfin, aucun autre caractère n'étant possible dans une expression arithmétique bien formée, on déclenche une erreur dans tout autre cas.

```
      | c -> failwith (Printf.sprintf "unknown character : %c" c)
```

La fonction auxiliaire `num` prend en paramètres un nombre `n` (formé par les chiffres déjà lus) et l'indice courant `i`. Elle parcourt l'entrée tant qu'elle n'y lit que des chiffres, et combine chaque chiffre lu avec son paramètre `n`. Elle renvoie le nombre final et l'indice courant lorsqu'elle atteint la fin de l'entrée ou lorsqu'elle lit un caractère autre qu'un chiffre.

```
  and num n i =
    if i >= String.length s then
      n, i
    else match s.[i] with
      | '0'..'9' as d ->
        num (10*n + Char.code d - Char.code '0') (i+1)
      | _ -> n, i
```

Pour analyser la chaîne complète, il ne reste plus qu'à lancer notre boucle principale à partir du premier caractère.

```
    in
    lex 0
```

Voici le code complet de l'analyseur.

```
let tokenize (s: string): token list =
  let rec lex i =
    if i >= String.length s then
      []
    else
      match s.[i] with
      | '+' -> PLUS :: (lex (i+1))
      | '*' -> STAR :: (lex (i+1))
      | '(' -> LPAR :: (lex (i+1))
      | ')' -> RPAR :: (lex (i+1))
      | ' ' -> lex (i+1)
      | '0'..'9' -> let n, i = num 0 i in
                    NUM n :: lex i
      | c -> failwith (Printf.sprintf "unknown character : %c" c)
  and num n i =
    if i >= String.length s then
      n, i
    else match s.[i] with
      | '0'..'9' as d ->
        num (10*n + Char.code d - Char.code '0') (i+1)
      | _ -> n, i
  in
  lex 0
```

Exercice 1.1. Étendre le type `token` et l'analyseur pour permettre la reconnaissance :

1. d'autres opérateurs arithmétiques : `-`, `/`, `mod`, `**` ;
2. de nombres négatifs ;
3. de nombres à virgule.

Attention aux multiples utilisations de certains caractères.

Exercice 1.2. Transformer la fonction récursive `lex` pour la rendre récursive terminale, et adapter la fonction `tokenizer`. *Attention à l'ordre des mots dans la liste renvoyée.*

Nous reviendrons sur l'analyse lexicale dans un chapitre dédié. Parmi les questions qui se poseront : comment décrire les mots d'un langage ? quels algorithmes pour les reconnaître efficacement ? quels sortes de mots peuvent ou non être décrits et reconnus ? Ce dernier point recèle des ramifications théoriques surprenantes.

1.3 Arbres de syntaxe et interprétation

Revenons sur la structure d'une expression arithmétique : il ne s'agit pas tant d'une séquence linéaire de mots que d'une structure hiérarchique où chaque opérateur est associé à deux opérandes. Cette structure est non seulement hiérarchique mais aussi récursive, puisque chaque opérande est à nouveau une expression arithmétique obéissant à la même structure.

C'est ainsi que l'on représente une expression arithmétique sous la forme d'un arbre appelé ***arbre de syntaxe abstraite***. Chaque nœud de l'arbre est un symbole choisi parmi un ensemble prédéfini (ici, des symboles pour l'addition, la multiplication et les constantes entières positives). Notez que chaque nœud portant un nombre entier est une feuille de l'arbre (un nœud avec zéro fils), et que chaque nœud portant un symbole d'addition ou de multiplication a exactement deux fils. On dit que l'addition et la multiplication ont une ***arité*** de deux, et la constante entière une arité de zéro.

On définit en caml un type de données pour représenter ces arbres de syntaxe de la manière suivante.

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

Notez que pour garder un ensemble fini de symboles, on a introduit un seul constructeur `Cst` pour les constantes entières, auquel on associe un nombre entier. Ainsi l'expression $1 + 2 \times 3$ est représentée en caml par la valeur `Add(Cst 1, Mul(Cst 2, Cst 3))` (de type `expr`).

On définit naturellement des fonctions manipulant des expressions arithmétiques sous la forme de fonctions récursives sur ce type `expr`. Voici par exemple des fonctions `nb_cst` et `nb_op` comptant respectivement le nombre de constantes entières et le nombre d'opérateurs dans une expression donnée en paramètre.

```
let rec nb_cst = function
  | Cst _ -> 1
  | Add(e1, e2) -> nb_cst e1 + nb_cst e2
  | Mul(e1, e2) -> nb_cst e1 + nb_cst e2

let rec nb_op = function
  | Cst _ -> 0
  | Add(e1, e2) -> 1 + nb_cst e1 + nb_cst e2
  | Mul(e1, e2) -> 1 + nb_cst e1 + nb_cst e2
```

On peut de même définir une fonction `eval` prenant en paramètre une expression arithmétique (ou plus précisément une valeur caml de type `expr` représentant cet expression arithmétique) et renvoyant le résultat du calcul décrit par cette expression. Cette fonction particulièrement simple fait correspondre chaque symbole à sa signification arithmétique après avoir évalué récursivement les sous-expressions.

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

Exercice 1.3. Étendre le type `expr` et la fonction `eval` pour intégrer les opérateurs arithmétiques et nombres de l'exercice 1.1.

La représentation arborescente des programmes sera la forme privilégiée, aussi bien pour les manipuler dans d'autres programmes comme la fonction `eval` *précédente, que pour les développements théoriques. Nous verrons d'ailleurs des techniques de preuve taillées sur mesure pour de telles structures arborescentes.*

1.4 Analyse grammaticale

Reste à savoir faire la transition entre la séquence de lexèmes produite par l'analyse lexicale et les arbres de syntaxe abstraite que l'on peut ensuite facilement manipuler à l'aide de fonctions récursives. Cette deuxième étape de l'analyse syntaxique, appelée ***analyse grammaticale*** doit regrouper les symboles formant ensemble des sous-expressions de l'expression principale, et combiner les sous-expressions avec les bons opérateurs.

On va utiliser ici l'algorithme de la gare de triage (*shunting yard*), qui lit la séquence de lexèmes de gauche à droite et stocke sur une pile les sous-expressions qui ont pu être formées avec les éléments déjà lus. Le principe est le suivant :

- les nombres sont directements transférés sur la pile des expressions, puisque chacun forme à lui seul une expression,
- les opérateurs sont placés en attente sur une pile auxiliaire d'opérateurs jusqu'à ce que leur opérande droit ait fini d'être analysé (comme on lit de gauche à droite, lorsque l'on rencontre un opérateur son opérande gauche a déjà été intégralement vu),
- lorsque l'on a au moins deux expressions et un opérateur au sommet de leurs piles respectives, ils 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.*

En outre, une parenthèse ouvrante est placée sur la pile auxiliaire des opérateurs en attendant que la parenthèse fermante associée soit vue.

L'algorithme s'arrête lorsque l'entrée a été complètement lue, que la pile auxiliaire des opérateurs est vide et qu'il ne reste plus qu'une expression sur la pile principale des expressions, cette dernière étant le résultat de l'analyse.

Dans le code ci-dessous, chaque pile est représentée à l'aide d'une simple liste. Lors de l'analyse de l'expression `2*3+4*5`, les trois piles ont successivement les états suivants (en écriture simplifiée).

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)
.	.	((4*5)+(2*3))

Dans l'état

entree	ops	exprs
+ 4 * 5	*	3 2

on a deux expressions dans `exprs` et un opérateur de multiplication dans `ops`. La multiplication étant l'opération la plus prioritaire, on forme l'expression `2*3`. Dans l'état

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

en revanche, l'opération d'addition qui serait possible pour former l'expression `(4+(2*3))` n'est pas utilisée, l'opérateur d'addition au sommet de `ops` étant moins prioritaire que l'opérateur de multiplication en tête de `entree`.

Voici un code caml pour cet algorithme, en représentant l'entrée et chacune des deux piles par une liste immuable. Notez que dans ce code, les premiers cas du **`match`** sont prioritaires sur les cas suivants.

```
let parse (s: string): expr =
 let rec sy (t: token list * token list * expr list) = match t with
   | ([], [], [e]) -> e

   | (NUM n :: input,  ops        ,  exprs              ) ->
   sy(input          ,  ops        ,  Cst n :: exprs     )

   | (LPAR  :: input,  ops        ,  exprs              ) ->
   sy(input          ,  LPAR :: ops,  exprs              )

   | (input          ,  STAR :: ops,  e1 :: e2 :: exprs  ) ->
   sy(input          ,  ops        ,  Mul(e1, e2) :: exprs)

   | (STAR  :: input,  ops        ,  exprs              ) ->
   sy(input          ,  STAR :: ops,  exprs              )

   | (input          ,  PLUS :: ops,  e1 :: e2 :: exprs  ) ->
   sy(input          ,  ops        ,  Add(e1, e2) :: exprs)

   | (PLUS  :: input,  ops        ,  exprs              ) ->
   sy(input          ,  PLUS :: ops,  exprs              )

   | (RPAR  :: input,  LPAR :: ops,  exprs              ) ->
   sy(input          ,  ops        ,  exprs              )

   | _ -> failwith "syntax error"
 in
 sy (tokenize s, [], [])
```

Exercice 1.4. Étendre l'analyseur pour permettre la reconnaissance des nouvelles opérations introduites à l'exercice 1.1, avec les bonnes priorités.
Attention aux opérateurs comme - et /, qui ne sont pas associatifs.

Nous avons considéré ici les règles traditionnelles d'écriture d'une expression mathématique. Nous verrons au cours de l'année comment décrire la syntaxe plus riche d'un langage de programmation, et les algorithmes et outils qui permettent de réaliser un analyseur syntaxique.

1.5 Analyse grammaticale, version structurée

Les algorithmes d'analyse grammaticale sont nombreux. Voici une deuxième approche, radicalement différente, basée sur la caractérisation suivante des expressions :

— 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.

On réalise un analyseur basé sur cette caractérisation à l'aide de trois fonctions mutuellement récursives `parse_expr`, `parse_m_expr` et `parse_atom`, chacune pour analyser l'une des trois formes possibles d'expressions, et deux fonctions auxiliaires `expand_sum` et `expand_prod`, respectivement pour gérer un enchaînement d'additions et un enchaînement de multiplications. Les trois fonctions `parse_*` prennent en paramètre une liste de lexèmes `l`, et renvoient d'une part l'expression construite et d'autre part la liste des lexèmes restants (type `token list -> expr * token list`). Les deux fonctions `expand_*` prennent en paramètre supplémentaire la partie de l'expression déjà construite (`expr * token list -> expr * token list`).

```
let rec parse_expr l = parse_m_expr l |> expand_sum
and expand_sum = function
  | e1, PLUS :: l -> let e2, l' = parse_m_expr l in
                     expand_sum (Add(e1, e2), l')
  | r -> r
and parse_m_expr l = parse_atom l |> expand_prod
and expand_prod = function
  | e1, STAR :: l -> let e2, l' = parse_atom l in
                     expand_prod (Mul(e1, e2), l')
  | r -> r
and parse_atom = function
  | NUM n :: l -> Cst n, l
  | LPAR :: l -> (match parse_expr l with
                  | e, RPAR :: l' -> e, l'
                  | _ -> failwith "syntax error")
  | _ -> failwith "syntax error"
```

Note : l'opérateur `|>` est une application inversée. On lit donc `parse_atom l |> expand_prod` comme « évaluer `parse_atom l`, puis appliquer la fonction `expand_prod` au résultat obtenu ».

Le point d'entrée de l'analyse est alors une fonction `parse`, qui appelle `parse_expr` et qui vérifie que tous les mots ont bien été consommés.

```
let parse (input: string): expr =
  match tokenize input |> parse_expr with
  | e, [] -> e
  | _ -> failwith "syntax error"
```

1.6 Compilation

Pour finir, nous allons traduire nos expressions arithmétiques (données par leur arbre de syntaxe abstraite) en séquences d'instructions élémentaires pour une machine virtuelle très simple stockant ses résultats intermédiaires sur une pile.

Imaginons donc une machine dotée de trois instructions seulement, représentées par le type caml suivant.

```
type instr =
  | ICst of int (* place une constante au sommet de la pile *)
  | IAdd        (* additionne deux valeurs au sommet de la pile *)
  | IMul        (* multiplie deux valeurs au sommet de la pile *)
```

La machine possède pour seule mémoire une pile. La valeur de la dernière expression évaluée est au sommet de la pile, et le reste de la pile est constitué de valeurs qui ont été mises en attente avant leur utilisation dans le calcul. L'instruction `ICst(n)` stocke l'entier `n` au sommet de la pile. Les instructions `IAdd` et `IMul` retirent les deux valeurs du sommet de la pile, effectuent respectivement leur addition ou leur multiplication, et placent le résultat au sommet de la pile.

Le résultat final de la machine est la valeur au sommet de la pile une fois toutes les instructions exécutées. Voici une fonction simulant l'exécution d'une telle machine (la pile est représentée ici par une liste).

```
let exec code =
  let rec ex (p: instr list * int list) = match p with
    | ([], 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, [])
```

Notre objectif de *compilation* consiste alors à traduire une expression `e` (de type `expr`) en une liste d'instructions `l` (de type `instr list`), de sorte que l'exécution de la liste d'instruction `l` produise la valeur de l'expression d'origine.

Pour cela, on traduit d'abord une expression constante par une simple instruction `ICst`. Pour une addition ou une multiplication, on produit d'abord une liste d'instructions calculant la valeur de l'opérande gauche (et la stockant au sommet de la pile), puis une liste d'instructions faisant de même avec l'opérande droit (qui vient donc au-dessus de la précédente sur la pile), et on termine par une instruction arithmétique combinant les deux valeurs obtenues.

```
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]
```

Exercice 1.5. Étendre le type `instr` des instructions de la machine virtuelle pour permettre le traitement des nouveaux éléments introduits à l'exercice 1.1. Étendre en conséquence les fonctions `exec` et `codegen`.

Exercice 1.6. Cette fonction `codegen` est terriblement inefficace, car elle utilise de manière intensive l'opérateur `@` de concaténation de listes, de complexité proportionnelle à la longueur de son premier opérande. Écrire une meilleure version, qui construit une liste élément par élément.

Dans la suite, plutôt que de cibler une machine à pile simpliste, nous générerons du code assembleur pour des ordinateurs réels. Cette partie s'interfacera avec des questions d'architecture et de système.

1.7 Question de style : tirer parti des impuretés

Le code caml vu jusqu'ici était purement fonctionnel, et ne manipulait que des données et des structures immuables. Cependant, le langage contient également des traits impératifs, et notamment des variables et des structures de données mutables. On tirera le meilleur parti du langage en mêlant ces deux aspects dans les bonnes proportions. Cette section illustre un tel mélange sur les programmes vus dans ce chapitre.

Syntaxe caml : références. Les variables mutables en caml sont appelées des ***références***. Le type `'a ref` désigne une référence contenant une valeur de type `'a`. On créer une référence en utilisant ce même mot-clé `ref`. Voici pour la définition d'une variable `count` désignant une référence entière (type `int ref`) initialisée avec la valeur `0`.

```
let count = ref 0 in ...
```

On accède à la valeur contenue dans une référence avec l'opérateur de *déréférencement* `!`, et on modifie la valeur d'une référence avec l'opérateur d'*affectation* `:=`. On incrémente de `1` la valeur de la référence `x` avec la ligne suivante.

```
x := !x + 1
```

L'opérateur de *séquence* ; sépare plusieurs telles opérations, à exécuter dans l'ordre. Ainsi

```
x := !x + 1;
x := 2 * !x;
x := !x + 3
```

va incrémenter la valeur de x de 1, puis la doubler, puis l'incrémenter à nouveau de 3, pour un résultat équivalent à ce qu'aurait donné `x := (!x + 1) * 2 + 3`. Notez l'absence de ; après la dernière affectation, qui n'a pas besoin d'être séparée de la suite.

À noter : contrairement à C, Java, Python et bien d'autres, Caml ne fait pas de distinction fondamentale entre les « expressions », qui calculent des valeurs, et les « instructions », qui produisent des actions. L'opération d'affectation `x := !x+1` n'est donc en caml qu'une expression ordinaire, produisant une valeur d'un type particulier `unit` (***unité***). Ce type contient une unique valeur, notée `()`, et caractérise les expressions dont la valeur n'est pas significative.

Ce type apparaît notamment dans les fonctions qui n'ont pas besoin de paramètre, ou encore dans les fonctions dont l'action consiste à produire un effet. Par exemple, la fonction `incr: int ref -> unit` s'applique à une référence entière et incrémente sa valeur de 1. On peut combiner tous les éléments précédents pour écrire une fonction `next: unit -> int` qui ne prend pas de paramètre (ou, plus précisément, qui accepte comme unique paramètre la valeur unité `()`), et qui produit un nouvel entier à chaque appel.

```
let counter = ref (-1)
let next () =
  incr counter;
  !counter
```

À noter au passage : l'opérateur de séquence ; sépare deux expressions. L'expression `e1; e2` évalue d'abord `e1`, dont on suppose qu'elle a le type `unit` [1], puis évalue `e2`. La valeur finale de `e1; e2` est celle de `e2`. La fonction `next` ci-dessus commence donc par incrémenter `counter`, puis renvoie la valeur lue après cet incrément.

Remarque : on peut même masquer la référence `counter` pour s'assurer que le reste du programme ne pourra pas interférer avec elle. Voici une telle variante.

```
let next =
  let counter = ref (-1) in          (* variable locale *)
  fun () -> incr counter; !counter   (* fonction associée à next *)
```

Génération de code instruction par instruction. Avec les éléments précédents, on peut écrire une fonction de génération de code qui agrandit petit à petit la liste d'instructions générées. On définit une référence `code` contenant une liste d'instructions (type `instr list ref`, à comprendre comme `(instr list) ref`) initialement vide, et une fonction auxiliaire `add_instr` qui ajoute une unique instruction à cette liste.

```
let codegen e =
  let code = ref [] in
  let add_instr i = code := i :: !code in
```

La fonction récursive `gen` se contente alors de parcourir l'expression et d'enregistrer chaque instruction émise avec `add_instr`.

```
  let rec gen: expr -> unit = function
    | Cst n       -> add_instr (ICst n)
    | Add(e1, e2) -> gen e1; gen e2; add_instr IAdd
    | Mul(e1, e2) -> gen e1; gen e2; add_instr IMul
  in
```

Ne reste à la fin qu'à appeler la fonction récursive `gen` sur l'expression complète et renvoyer la liste accumulée dans `code`. Attention à un point d'ordre : comme `add_instr` ajoute chaque nouvelle instruction *en tête* de la liste, il faut penser à renverser la liste obtenue.

```
  gen e;
  List.rev !code
```

1. Dans le cas où `e1` n'est pas du type `unit`, le compilateur émet un avertissement, mais pas une erreur.

Analyse lexicale incrémentale. Avec les éléments précédents, on peut écrire un analyseur lexical qui ne produit pas d'un coup la liste intégrale de tous les mots de l'entrée (à l'échelle d'un programme entier, cela peut être long et inutilement coûteux en mémoire), mais qui fournit chaque mot « à la demande ».

Pour fluidifier l'utilisation d'une telle fonction d'analyse, on étend le type des mots d'un symbole spécial `EOI` (*End Of Input*), à renvoyer lorsqu'un utilisateur demande le prochain mot d'une entrée qui a déjà été parcourue intégralement.

```
type token =
  | NUM of int
  | PLUS
  | STAR
  | LPAR
  | RPAR
  | EOI
```

La fonction `tokenizer` prend en paramètre la chaîne `s` à analyser, et renvoie une fonction `next_token: unit -> token` qui, à chaque nouvel appel, renvoie le prochain mot. Pour cela, on utilise une référence `i` qui mémorise la position courante dans la chaîne `s`, et qui est incrémentée à mesure des appels à `next_token`.

```
let tokenizer (s: string): unit -> token =
  let i = ref 0 in
  let rec next_token () =
    if !i >= String.length s then
      EOI
    else
      match s.[!i] with
      | '+' -> incr i; PLUS
      | '*' -> incr i; STAR
      | '(' -> incr i; LPAR
      | ')' -> incr i; RPAR
      | ' ' -> incr i; next_token()
      | '0'..'9' -> NUM (next_num 0)
      | c -> failwith (Printf.sprintf "unknown character : %c" c)
  and next_num n =
    if !i >= String.length s then
      n
    else match s.[!i] with
      | '0'..'9' as d ->
        incr i; next_num (10*n + Char.code d - Char.code '0')
      | _ -> n
  in
  next_token
```

Remarque : contrairement à leurs équivalents purement fonctionnels `token` et `num`, les fonctions `next_token` et `next_num` n'ont plus besoin de prendre en paramètre la position `i` à lire, ni de renvoyer la position atteinte après lecture, car la référence `i` contient cette information. Notez qu'à chaque fois qu'un caractère est consommé, on utilise la fonction `incr` pour mettre `i` à jour.

L'analyse grammaticale peut ensuite être elle-même adaptée à ce nouveau style. Voici pour la version récursive de l'analyse. On utilise la fonction `tokenizer` pour initialiser notre générateur de mots, et on crée une référence `current_token` mémorisant le mot courant (ou `EOI` si l'entrée a déjà été intégralement consommée). Une fonction `next` met à jour le mot courant à l'aide d'un nouvel appel à `next_token`. La présence de cette référence `current_token` dispense de mentionner la liste des mots dans les paramètres ou les résultats des fonctions d'analyse (même phénomène que pour le compteur `i` dans `tokenizer`). Il suffit d'appeler `next` à chaque fois qu'un mot est effectivement consommé.

```
let parse (s: string): expr =
  let next_token = tokenizer s in
  let current_token = ref (next_token()) in
  let next() = current_token := next_token() in

  let rec parse_expr() = parse_m_expr() |> expand_sum
  and expand_sum e = match !current_token with
```

```
    | PLUS -> next(); Add(e, parse_m_expr()) |> expand_sum
    | _ -> e
  and parse_m_expr() = parse_atom() |> expand_prod
  and expand_prod e = match !current_token with
    | STAR -> next(); Mul(e, parse_atom()) |> expand_prod
    | _ -> e
  and parse_atom() = match !current_token with
    | NUM n -> next(); Cst n
    | LPAR  -> next(); let e = parse_expr() in
               (match !current_token with
                | RPAR -> next(); e
                | _ -> failwith "syntax error")
    | _ -> failwith "syntax error"
  in
  if !current_token <> EOI then failwith "syntax error";
  parse_expr()
```

Bibliothèque caml : piles mutables. La bibliothèque standard de caml offre un certain nombre de structures de données mutables : tableaux, piles, files, tables de hachage... On va utiliser ici le module `Stack` des piles (LIFO) mutables. Ce module définit notamment

- le type `'a Stack.t` des piles contenant des éléments de type `'a`,
- une fonction `Stack.create: unit -> 'a Stack.t` de création d'une pile vide,
- une fonction `Stack.push: 'a -> 'a Stack.t -> unit` qui ajoute un élément au sommet d'une pile,
- une fonction `Stack.pop: 'a Stack.t -> 'a` qui retire l'élément au sommet d'une pile et le renvoie (erreur si pile vide),
- une fonction `Stack.is_empty: 'a Stack.t -> bool` qui teste si une pile est vide.

On peut utiliser cette structure pour représenter la pile manipulée par notre fonction `exec`. La fonction d'exécution commence donc par initialiser une pile vide `stack`, dans laquelle seront sauvegardées les valeurs intermédiaires du calcul.

```
let exec code =
  let stack = Stack.create() in
```

L'exécution elle-même consiste alors, pour chaque instruction prise dans l'ordre, à effectuer la bonne combinaison d'ajout ou de retrait d'éléments dans la pile.

```
  let open Stack in
  List.iter (function
      | ICst n -> push n stack
      | IAdd   -> let v1, v2 = pop stack, pop stack in
                  push (v1+v2) stack
      | IMul   -> let v1, v2 = pop stack, pop stack in
                  push (v1*v2) stack
    ) code;
```

Point de syntaxe : la ligne `let open Stack in` est une ouverture locale de module. Dans l'expression qui suit cette ouverture locale, on peut accéder aux fonctions du module `Stack` sans utiliser le préfixe `Stack.`. Une fois l'exécution terminée, on conclut en récupérant l'élément au sommet de la pile (normalement, il est seul sur la pile).

```
  let v = pop stack in
  assert (is_empty stack);
  v
```

Pour une dernière amélioration, remarquez que le dernier élément ajouté sur la pile est toujours systématiquement retiré immédiatement après. On peut couper court en gardant à part la dernière valeur calculée et en la transmettant directement, sans passer par la pile.

```
let exec code =
  let stack = Stack.create() in
  List.fold_left (fun v -> function
      | ICst n -> Stack.push reg stack; n
      | IAdd   -> Stack.pop stack + v
      | IMul   -> Stack.pop stack * v
    ) 0 code
```

2 Interprétation d'un langage impératif

Dans ce chapitre, on définit et on interprète un langage impératif minimal, nommé IMP. Voici un exemple de programme 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);
```

Avez-vous reconnu cet algorithme ?

On distingue dans ce langage une notion d'expression, formée de constantes entières, de variables, d'opérations arithmétiques et logiques, et une notion d'instruction comprenant l'affectation d'une valeur à une variable, la boucle `while` et le branchement conditionnel `if`/`else`. L'instruction `print` affiche un caractère donné par son code ASCII.

Au-delà des détails de syntaxe ou d'implémentation, on va s'intéresser à ce qui définit le cœur d'un tel langage de programmation. Un outil central du chapitre est l'***induction structurelle***, aussi appelée *récurrence structurelle*, qui permet de définir, manipuler, raisonner sur et programmer des structures intrinsèquement récursives comme peut l'être un programme structuré.

2.1 Syntaxe concrète et syntaxe abstraite

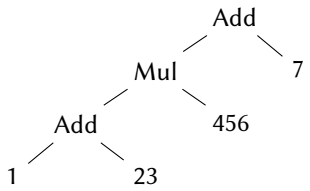
Notre calculatrice du chapitre précédent permettait d'écrire des calculs variées. Par exemple :

```
> (1+23)*456+7
> (1 + 23) * 456 + 7
> ( 1+23 ) *456   +7
```

Ces trois écritures ont beau être différentes, elle représentent toutes la même expression arithmétique $(1 + 23) \times 456 + 7$.

Ainsi, on distingue deux niveaux de syntaxe pour un langage de programmation. La ***syntaxe concrète*** correspond à ce que le programmeur écrit. Il s'agit d'un texte, une chaîne de caractères. À l'inverse, la ***syntaxe abstraite*** donne une représentation structurée du programme, sous la forme d'un arbre. La syntaxe concrète représente donc la partie visible du langage. La syntaxe abstraite est un outil central à la fois pour manipuler les programmes à l'intérieur du compilateur et pour raisonneer formellement sur les programmes. En outre, la syntaxe abstraite tend à être plus centrée sur le cœur du langage et à s'abstraire de certains points annexes de l'écriture.

Différences entre les deux niveaux de syntaxe D'une part, certains éléments non-signifiants de la syntaxe concrète, comme les espaces ou les commentaires, n'apparaissent plus dans la représentation en syntaxe abstraite. De même, les parenthèses qui servaient à correctement associer les opérateurs à leurs opérandes ne sont plus utiles. Ainsi, les trois chaînes de caractères précédentes sont représentées par le même arbre de syntaxe abstraite arithmétique



D'autre part, la syntaxe concrète des langages de programmation offre souvent des écritures simplifiées, appelées *sucre syntaxique*, pour certaines opérations courantes. Ces formes simplifiées cependant ne viennent pas ajouter de nouvelles structures à la syntaxe

abstraite, et sont simplement comprises comme une combinaison de certains éléments déjà présents dans le cœur du langage. Par exemple :

- en python l'instruction d'incrément `x += 1` est un raccourci d'écriture pour la simple instruction d'affectation `x = x + 1` et produit le même arbre de syntaxe abstraite ;
- en C, l'expression d'accès à une case de tableau `t[i]` n'est en réalité rien d'autre qu'une petite expression `*(t+i)` d'arithmétique de pointeurs [2] ;
- en caml enfin, la définition d'une fonction avec **`let`** `f x = e` est en réalité une définition de variable ordinaire, à laquelle on affecte une fonction anonyme : **`let`** `f =` **`fun`** `x -> e`, et de même la définition d'une fonction à deux paramètres **`let`** `f x y = e` se décompose en **`let`** `f =` **`fun`** `x ->` **`fun`** `y -> e`.

2.2 Structures inductives

Les arbres de syntaxe abstraite des programmes ont une structure ***inductive***, que l'on peut décrire à l'aide d'une forme de récurrence : un programme est construit en combinant plusieurs fragments de programmes, eux-mêmes obtenus par combinaisons d'éléments plus petits, etc. Cette structure permet de facilement définir des *fonctions récursives* s'appliquant à des arbres de syntaxe abstraite, ou encore de raisonner sur les programmes à l'aide du *principe d'induction structurelle*.

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 : 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.

Exemple : expressions arithmétiques. On peut définir un 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.

On peut alléger la manipulation de telles expressions à l'aide de notations mathématiques comme n, $e_1 \oplus e_2$ et $e_1 \otimes e_2$. Dans ce chapitre, on garde cependant des symboles $\oplus$ et $\otimes$ bien distincts des opérateurs $+$ et $\times$ classiques, pour distinguer le langage et son interprétation.

- L'opérateur $+$ est un élément mathématique, qui s'applique à deux nombres pour en calculer un troisième. Il vérifie l'équation $1 + 2 = 3$ et servira à interpréter notre langage d'expressions.
- 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 $1 \oplus 2 \neq 3$.

Attention également à toujours utiliser cette notation allégée avec suffisamment de parenthèses pour éviter toute ambiguïté sur la structure des expressions. On s'interdira donc d'écrire $e_1 \oplus e_2 \oplus e_3 \oplus e_4$ et on choisira à la place l'une des cinq structures possibles, comme

2. Ce sucre syntaxique a des conséquences amusantes, puisqu'il permet d'écrire `2[t]` pour obtenir le même résultat que `t[2]`.

par exemple $(e_1 \oplus e_2) \oplus (e_3 \oplus e_4)$ ou $e_1 \oplus ((e_2 \oplus e_3) \oplus e_4)$. On évitera même de se reposer sur les conventions mathématiques de priorité : on utilisera le parenthésage explicite $1 \oplus (2 \otimes 3)$ pour représenter l'expression mathématique usuelle $1 + 2 \times 3$.

Définition de fonctions sur un ensemble d'objets inductifs. 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 $f(c(e_1, \dots, e_n))$ en fonction des sous-éléments e_i et de leurs images $f(e_i)$ par la fonction.

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

Voici par exemple trois ensembles d'équations sur nos termes représentant des expressions arithmétiques. Ces équations définissent des fonctions nbCst, nbOp et eval, telles que nbCst(e) donne le nombre de constantes dans l'expression arithmétique e, nbOp(e) le nombre d'opérateurs apparaissant dans e, et eval(e) donne la valeur obtenue en effectuant le calcul décrit par e. Remarquez que l'écriture de telles fonctions demande de bien distinguer les expressions arithmétiques elles-mêmes (la *syntaxe*) de la valeur associé (la *sémantique*). En particulier, le constructeur $\oplus$ est un élément syntaxique représentant une addition (un constructeur), à ne pas confondre avec l'opérateur $+$ qui est l'interprétation mathématique de l'addition (une fonction).

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

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

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

Programmer avec des objets inductifs. Les types algébriques de caml permettent précisément de définir des ensemble d'objets inductifs, en fournissant un ensemble de constructeurs, et pour chaque constructeur les types des éléments qu'il combine. Note : un objet de base est finalement vu ici comme un constructeur d'arité zéro.

Ainsi, les listes contenant des éléments de type `'a` sont définies par l'objet de base `[]`, et par un constructeur `::` s'appliquant à un élément et à une liste.

```
type 'a list =
  | []
  | (::) of 'a * 'a list
```

On peut de même définir des expressions à l'aide de trois constructeurs.

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

Note : on ne peut pas définir en caml une infinité d'objets de base, on utilise donc pour les constantes entières un unique constructeur `Cst`, paramétré par un entier. Avec une telle définition, on pourra représenter l'expression $(1 \oplus 2) \oplus (3 \otimes 4)$ en caml par le code :

```
Add(Add(Cst 1, Cst 2), Mul(Cst 3, Cst 4))
```

Avec de tels types algébriques, les équations utilisées ci-dessus pour définir des fonctions sur les expressions arithmétiques se traduisent directement en code, sous forme de fonctions récursives.

```
let rec nb_cst = function
  | Cst n       -> 1
```

```
    | Add(e1, e2) -> nb_cst e1 + nb_cst e2
    | Mul(e1, e2) -> nb_cst e1 + nb_cst e2

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

À noter également : on peut ignorer les parties de l'expression qui ne sont pas utilisées dans le calcul, et regrouper les cas identiques.

```
  let rec nb_op = function
    | Cst _                      -> 0
    | Add(e1, e2) | Mul(e1, e2) -> nb_op e1 + nb_op e2
```

Raisonnement par induction structurelle. Dans un ensemble E d'objets inductifs, chaque objet peut être construit en utilisant uniquement les objets de base et les constructeurs. Pour démontrer qu'une propriété P est valide 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 vérifiant tous $P(e_i)$, alors la propriété P vaut encore pour l'élément construit $c(e_1, \dots, e_n)$. Autrement dit, $P(e_1) \wedge \dots \wedge P(e_n) \implies P(c(e_1, \dots, e_n))$.

Ainsi, on assure qu'il impossible de construire un élément e qui ne vérifierait pas la propriété [3].

Cette technique de preuve est une ***preuve par récurrence***, qu'on appelle plus précisément *preuve par récurrence structurelle*, ou encore ***preuve par induction structurelle***. Le premier point décrit ses ***cas de base*** (un par objet de base) et le deuxième point ses ***cas récursifs*** (un par moyen de combiner des objets). Dans le deuxième point, les hypothèses $P(e_1)$ à $P(e_n)$ que l'on peut utiliser pour justifier $P(c(t_1, \dots, t_n))$ sont les *hypothèses de récurrence*, ou ***hypothèse d'induction***.

Voici comment appliquer ce principe à nos deux exemples.

- Pour démontrer qu'une propriété P est vraie pour toutes les listes, il suffit de démontrer que :
  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 $e :: \ell$.
- Pour démontrer qu'une propriété P est vraie pour toutes les expressions arithmétiques, il suffit de démontrer que :
  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 $e_1 \otimes e_2$.

Montrons par exemple que dans toute expression arithmétique, le nombre de constantes est de un supérieur au nombre d'opérateurs. Pour cela, notons $P(e)$ la propriété $\mathsf{nbCst}(e) = \mathsf{nbOp}(e) + 1$, et vérifions les cas de base et les cas récursifs correspondant aux différents symboles de la signature :

- Cas de la constante (cas de base) : pour tout terme constant n on a $\mathsf{nbCst}(\mathsf{n}) = 1$ et $\mathsf{nbOp}(\mathsf{n}) = 0$. Ainsi 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

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

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

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

3. Notez à quel point ce paragraphe est parallèle à celui concernant la définition d'une fonction !

2.3 Variables et environnements

Notez la différence avec la notion de variable en mathématiques, qui désigne au contraire une chose indéterminée.

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
```

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 change cependant de nature : elle a maintenant besoin d'une information sur la mémoire, ou plus précisément sur les valeurs associées aux différentes variables. On peut abstraire cette information sous la forme d'une fonction ρ appelée ***environnement***, qui à chaque nom de variables associe sa valeur.

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

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

L'environnement ρ évolue à mesure que l'évaluation d'un programme progresse, mais la nature de cette évolution peut être très différente d'un paradigme de programmation à l'autre. Pour l'instant, on se concentre sur la version fonctionnelle, dont la description théorique est plus simple.

Version fonctionnelle : variables locales immuables. Dans un langage comme caml, les variables sont ***immuables*** : elles reçoivent lors de leur déclaration une valeur, qui n'est jamais modifiée.

L'expression `let y = 1+2*x in 2*x*y` définit une variable `y` en lui donnant la valeur calculée par l'expression `1+2*x`. Cette valeur peut ensuite être utilisée lors de l'évaluation de l'expression `2*x*y`. Autrement dit, l'expression `1+2*x` est évaluée dans un certain environnement ρ, donnant notamment la valeur de `x`, 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`.

Notez que cette variable est locale à l'expression `2*x*y` située à droite du **`in`**, elle n'existe pas dans les autres parties du programme. Autrement dit, 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 $y \neq x$. On peut alors résumer le principe d'évaluation d'une expression let par l'équation suivante.

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

Représentation en caml. On peut étendre notre 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
```

Alors 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 On se donne pour représenter les valeurs produites par les expressions arithmétiques un type `value` désignant simplement des nombres entiers.

```
type value = int
```

Un environnement doit associer des noms de variables (c'est-à-dire des chaînes de caractères) à des valeurs (de type `value`). On a donc besoin d'une structure de ***table associative***, qui peut être réalisée à l'aide de différentes structures de données. Citons notamment :

— les arbres de recherche équilibrés (module `Map` de caml), qui permettent de représenter un environnement comme une structure de données immuable, en style purement fonctionnel,

— les tables de hachage (module `Hashtbl` de caml), qui forment une structure de données mutable.

On va utiliser ici la solution à base d'arbres de recherche équilibrés, que l'on peut introduire par les déclarations suivantes.

```
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 et de 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.

Notez qu'en pratique, lorsque les approches à base d'arbres de recherche et à base de tables de hachage sont toutes deux possibles, la deuxième est (un peu) plus efficace. Les accès sont en effet en temps constant plutôt que logarithmique. Ici, nous utilisons la version purement fonctionnelle à base d'arbres de recherche, car elle est plus facile à mettre en œuvre.

Écriture d'un interprète Maintenant que l'on a mis en place les éléments pour représenter une expression (à l'aide de sa syntaxe abstraite) et un environnement (à l'aide d'une table associative), on peut écrire une fonction caml

```
eval: expr -> env -> value
```

telle que `eval` e ρ renvoie la valeur de l'expression e évaluée dans l'environnement ρ. Il suffit pour cela d'écrire un cas pour chacune des équations déjà vues pour eval.

```
let rec eval e env = match e with
  | Cst n -> n
  | Add(e1, e2) -> eval e1 env + eval e2 env
  | Mul(e1, e2) -> eval e1 env * eval e2 env
```

L'accès à une variable consulte l'environnement. La déclaration d'une variable locale avec `let x = e1 in e2` définit un environnement étendu associant `x` à la valeur de `e1`, et évalue l'expression `e2` dans ce nouvel environnement.

```
  | Var x -> Env.find x env
  | Let(x, e1, e2) ->
     let v1 = eval e1 env in
     let env' = Env.add x v1 env in
     eval e2 env'
```

On en déduit une fonction `eval_top` procédant à l'évaluation d'une expression dans l'environnement vide.

```
let eval_top (e: expr): value =
  eval e Env.empty
```

Exercice 2.1. Écrire un interprète pour les expressions précédentes, en utilisant une table de hachage plutôt qu'une structure `Map`. Attention, il y a un piège.

2.4 Variables mutables et instructions

Nous avons dit précédemment : 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
```

Nous avons vu jusqu'ici le 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.

L'instruction `y = 1+2*x` affecte à la variable `y` une nouvelle valeur, calculée par l'expression `1+2*x`. Une fois cette instruction exécutée dans un certain environnement ρ, toute la suite du programme sera exécutée dans l'environnement modifié ρ', qui est identique à ρ si ce n'est qu'il associe à y la nouvelle valeur.

Instructions. En plus des expressions, on définit donc un nouvel ensemble d'objets inductifs représentant les ***instructions***. On prend pour l'instant une unique construction binaire `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

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

Dans un tel langage, toute nouvelle affectation écrase la valeur précédente de la variable. Ainsi dans le programme

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

les deux occurrences du nom `x` désignent bien la même variable. La deuxième affectation remplace définitivement l'association de `x` à 1 par une association de `x` à 2.

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;
```

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

Représentation en caml et interprétation. On peut se donner des définitions caml minimales pour des expressions arithmétiques avec variables, et des séquences d'instructions d'affectation. On conserve les notions de valeurs et d'environnement déjà utilisées, on garde l'essentiel de la définition des expressions simples, et on n'ajoute que la notion d'instruction.

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

type expr =
  | Cst of int
  | Var of string
  | Add of expr * expr
  | Mul of expr * expr

type instr =
  | Set of string * expr
```

La fonction `eval: expr -> env -> value` peut être définie comme on l'a déjà fait pour les variables immuables, et on ajoute ici deux fonctions `exec: instr -> env -> env` et `exec_seq: instr list -> env -> env` respectivement pour l'exécution d'une instruction et pour l'exécution d'une séquence. L'une et l'autre renvoient comme résultat un nouvel environnement tenant compte des affectations effectuées.

```
let rec eval e env = match e with
  | Cst n -> n
  | Var x -> Env.find x env
  | Add(e1, e2) -> eval e1 env + eval e2 env
  | Mul(e1, e2) -> eval e1 env * eval e2 env

let exec_instr i env = match i with
  | Set(x, e) -> let v = eval e env in
                 Env.add x v env

let rec exec_seq s env = match s with
  | []      -> env
  | i :: s' -> let env' = exec_instr i env in
               exec_seq s' env'
```

Remarquez que l'on vient d'écrire une fonction d'interprétation pour des instructions manipulant des variables mutables, à l'aide uniquement d'une structure de données immuable. C'est tout à fait possible ! Et c'est au plus proche des équations mathématiques écrites à la page précédente.

Si vous aimez les itérateurs en caml, vous pouvez remplacer la définition de `exec_seq` par un appel à `List.fold_left`.

Variante avec structure de données mutable. Bien que ce ne soit techniquement pas indispensable et que, à terme, cela complexifie le raisonnement sur les programmes, on peut réaliser une variante des fonctions `exec` précédentes à l'aide d'une structure de données mutable comme une table de hachage.

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

La fonction d'évaluation est quasiment identique, mais utilise cette fois la fonction `find` de la bibliothèque `Hashtbl` (petit détail historique : cette fonction prend ses arguments dans l'ordre inverse par rapport à celle de `Map`).

```
let rec eval (e: expr) (env: env): value = match e with
  | Cst n -> n
  | Var x -> Hashtbl.find env x
  | Add(e1, e2) -> eval e1 env + eval e2 env
  | Mul(e1, e2) -> eval e1 env * eval e2 env
```

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` qui crée une nouvelle association entre une clé et une valeur, ou remplace l'association précédente si la clé existait déjà. On a donc les nouveaux types `exec_instr: instr -> env -> unit` et `exec_seq: instr list -> env -> unit`.

```
let exec_instr i env = match i with
  | Set(x, e) -> let v = eval e env in
                 Hashtbl.replace env x v

let rec exec_seq s env = match s with
  | []      -> ()
  | i :: s' -> exec_instr i env; exec_seq s' env
```

On charge ensuite la fonction principale `exec_top` de créer une table vide dans laquelle commencer l'exécution. Ici on demande à cette dernière de renvoyer la table à la fin, pour pouvoir consulter son état.

```
let exec_top (s: instr list): env =
  let env = Hashtbl.create 64 in
  exec_seq s env;
  env
```

Cette nouvelle fonction d'interprétation respecte toujours, d'une certaine manière, les équations de sémantique, mais du fait de la mutation de la table de hachage cette correspondance n'est plus si immédiate. Avec les équations de sémantique et l'interprète purement fonctionnel, on pouvait écrire des égalités de la forme

$$\text{exec}(s, \rho) = \rho'$$

liant deux environnements ρ et ρ' distincts. Avec les tables de hachage en revanche, on n'a plus qu'un unique environnement, passant par plusieurs états. On retrouve un lien avec les équations précédentes en distinguant un unique environnement `env` (une structure de données) et ses états successifs ρ_1, ρ_2, ρ_3, ... (différentes fonctions des variables vers les valeurs). Dans l'interprète impératif, les équations s'appliquent exclusivement aux états ρ_i de la structure mutable `env`, et pas à la structure elle-même. L'équation $\text{exec}(s, \rho) = \rho'$ se traduit alors par le fait que l'exécution de la séquence `s` fait passer l'environnement `env` de l'état ρ à l'état ρ'.

Variante compacte du code. Comme l'exécution d'un programme utilise une unique table de hachage, on peut encore alléger le code en faisant de `env` une variable à laquelle toutes les fonctions auront accès, sans qu'il soit besoin de la passer en paramètre à chaque appel. Pour cela, on fait de `eval`, `exec_instr` et `exec_seq` des fonctions locales. Au passage, on remplace `exec_seq` par un itérateur.

```
let exec (s: instr list): env =
  let env = Hashtbl.create 64 in
  let rec eval = function
    | Cst n -> n
    | Var x -> Hashtbl.find env x
    | Add(e1, e2) -> eval e1 + eval e2
    | Mul(e1, e2) -> eval e1 * eval e2
  in
  let rec exec_instr = function
    | Set(x, e) -> let v = eval e in
                   Hashtbl.replace env x v
  in
  let exec_seq = List.iter exec_instr in
  exec_seq s; env
```

2.5 Approfondissement : interprète complet pour IMP

Considérons maintenant notre mini-langage IMP, avec arithmétique, variables mutables, et instructions structurées.

```
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);
```

Syntaxe abstraite. La syntaxe abstraite de ce langage est donnée principalement par un ensemble d'expressions et un ensemble d'instructions. Voici une définition en caml pour les expressions, avec un constructeur pour les constantes entières, un pour les variables, et un constructeur `Bop` regroupant un certain nombre d'opérations binaires : `Add` pour l'addition `+`, `Mul` pour la multiplication `*`, `Lt` pour la comparaison `<`, `And` pour la conjonction `&&`...

```
type bop = Add | Mul | Lt | And | ...
type expr =
  | Cst  of int
  | Var  of string
  | Bop  of bop * expr * expr
```

Pour les instructions on se donne : un constructeur `Set` pour l'affectation, des constructeurs `If` et `While` pour les branchements et les boucles, et un constructeur `Print` pour une instruction affichant un caractère sur la sortie standard.

```
type instr =
  | Set   of string * expr
  | If    of expr * seq * seq
  | While of expr * seq
  | Print of expr
and seq = instr list
```

La séquence d'instructions IMP donnée plus haut est ainsi représentée en ocaml par la liste

```
[ Set("a", Cst 2)
; Set("n", Cst 6)
; Set("r", Cst 1)
; While(Bop(Lt, Cst 0, Var "n"),
        [ If(Bop(Eq, Bop(Mod, Var "n", Cst 2), Cst 1),
             [ Set("r", Bop(Mul, Var "r", Var "a")) ],
             [])
        ; Set("a", Bop(Mul, Var "a", Var "a"))
        ; Set("n", Bop(Div, Var "n", Cst 2  )) ])
; Print(Var "r") ]
```

Interprétation. On va reprendre le principe de l'évaluateur utilisant une table de hachage pour l'environnement mutable, avec une notation un peu plus compacte. Pour cela, remarquons que toutes les fonctions `eval`, `exec` et `exec_seq` partagent une unique table `env`. On peut donc définir cette table une fois pour toute à l'extérieur, puis définir nos trois fonctions sans qu'elles aient besoin de prendre l'environnement en paramètre. La fonction `eval`, par exemple, n'a plus besoin de prendre comme argument que l'expression à évaluer, et on peut ensuite conserver les règles déjà données pour les expressions.

```
let exec_prog (p: seq): unit =
  let env = Hashtbl.create 64 in

  let rec eval: expr -> value = function
    | Cst n -> n
    | Var x -> Hashtbl.find env x
    | Bop(Add, e1, e2) -> eval e1 + eval e2
    | Bop(Mul, e1, e2) -> eval e1 * eval e2
```

On a comme nouveauté une décision à prendre quant à la valeur produite par le nouvel opérateur de comparaison. On peut par exemple introduire des valeurs particulières vrai et faux à côté des nombres déjà utilisés, ou encore réutiliser des entiers particuliers, par exemple 1 pour vrai et 0 pour faux.

$$\text{eval}(e_1 \lessgtr e_2, \rho) \quad = \quad \begin{cases} 1 & \text{si eval}(e_1,\rho) < \text{eval}(e_2,\rho) \\ 0 & \text{sinon} \end{cases}$$

On donne également à la conjonction sa sémantique paresseuse, en ne faisant intervenir l'opérande de droite que lorsque cela est nécessaire.

```
    | Bop(Lt,  e1, e2) -> if eval e1 < eval e2 then 1 else 0
    | Bop(And, e1, e2) -> if eval e1 = 0 then 0 else eval e2
```

On reprend de même les équations déjà données pour la fonction exec, en l'étendant pour tenir compte des constructions **`if`** et **`while`**. Dans chaque cas on obtient une équation avec deux voies possibles, en fonction de la valeur obtenue en évaluant la garde.

$$\text{exec}(\texttt{if (}e\texttt{) \{}b_1\texttt{\} else \{}b_2\texttt{\}}, \rho) \quad = \quad \begin{cases} \text{exec}(b_1,\rho) & \text{si eval}(e,\rho) \neq 0 \\ \text{exec}(b_2,\rho) & \text{sinon} \end{cases}$$

$$\text{exec}(\texttt{while (}e\texttt{) \{}b\texttt{\}}, \rho) \quad = \quad \begin{cases} \rho & \text{si eval}(e,\rho) = 0 \\ \text{exec}(\texttt{while (}e\texttt{) \{}b\texttt{\}}, \text{exec}(b,\rho)) & \text{sinon} \end{cases}$$

Pour exécuter une séquence d'instructions, il suffit d'itérer la fonction `exec`.

```
  and exec: instr -> unit = function
    | Set(x, e) -> let v = eval e in Hashtbl.replace env x v
    | If(e, s1, s2) ->
       if eval e <> 0 then exec_seq s1 else exec_seq s2
    | While(e, s) as i ->
       if eval e <> 0 then (exec_seq s; exec i)
    | Print(e) -> let v = eval e in Printf.printf "%c\n" (Char.chr v)
  and exec_seq (s: seq): unit =
    List.iter exec s
  in
  exec_seq p
```

Voici quelques étapes de l'exécution de notre programme exemple, en partant de l'environement vide. On a d'abord une phase initialisant les variables `n` et `r`, puis une succession d'étapes dans la boucle **`while`**.

$$
\begin{aligned}
&\texttt{exec}(\texttt{n := 6;r := 1;while...}, \emptyset)\\
=\ &\texttt{exec}(\texttt{while...}, \texttt{exec}(\texttt{r := 1}, \texttt{exec}(\texttt{n := 6}, \emptyset)))\\
=\ &\texttt{exec}(\texttt{while...}, [n \mapsto 6, r \mapsto 1])\\
=\ &\texttt{exec}(\texttt{while...}, \texttt{exec}(\texttt{r := } r \otimes n\texttt{;n := } n \oplus (-1), [n \mapsto 6, r \mapsto 1]))\\
&\qquad \text{car} \left| \begin{array}{l} \text{eval}(0 \lessdot n, [n \mapsto 6, r \mapsto 1]) \ = \ 1 \\ \quad \text{car} \left| \begin{array}{l} \text{eval}(0, [n \mapsto 6, r \mapsto 1]) \ = \ 0 \\ \qquad\qquad < \ 6 \ = \ \text{eval}(n, [n \mapsto 6, r \mapsto 1]) \end{array} \right. \end{array} \right.\\
=\ &\texttt{exec}(\texttt{while...}, [n \mapsto 5, r \mapsto 6])\\
=\ &\texttt{exec}(\texttt{while...}, [n \mapsto 4, r \mapsto 30])\\
=\ &\texttt{exec}(\texttt{while...}, [n \mapsto 3, r \mapsto 120])\\
=\ &\texttt{exec}(\texttt{while...}, [n \mapsto 2, r \mapsto 360])\\
=\ &\texttt{exec}(\texttt{while...}, [n \mapsto 1, r \mapsto 720])\\
=\ &\texttt{exec}(\texttt{while...}, [n \mapsto 0, r \mapsto 720])\\
=\ &[n \mapsto 0, r \mapsto 720]\\
&\qquad \text{car} \left| \begin{array}{l} \text{eval}(0 \lessdot n, [n \mapsto 0, r \mapsto 720]) \ = \ 0 \\ \quad \text{car} \left| \begin{array}{l} \text{eval}(0, [n \mapsto 0, r \mapsto 720]) \ = \ 0 \\ \qquad\qquad \not< \ 0 \ = \ \text{eval}(n, [n \mapsto 0, r \mapsto 720]) \end{array} \right. \end{array} \right.
\end{aligned}
$$

Si l'on tentait de faire un tel raisonnement sur le programme `x := 0;while (1) {x :=` $x \oplus 1$`}`, le calcul aurait une forme différente.

$$
\begin{aligned}
&\texttt{exec}(\texttt{x := 0;while (1) \{x :=}\ x \oplus 1\texttt{\}}, \emptyset)\\
=\ &\texttt{exec}(\texttt{while (1) \{x :=}\ x \oplus 1\texttt{\}}, [x \mapsto 0])\\
&\quad \text{car eval}(1, [x \mapsto 0]) = 1\\
=\ &\texttt{exec}(\texttt{while (1) \{x :=}\ x \oplus 1\texttt{\}}, [x \mapsto 1])\\
&\quad \text{car eval}(1, [x \mapsto 1]) = 1\\
=\ &\texttt{exec}(\texttt{while (1) \{x :=}\ x \oplus 1\texttt{\}}, [x \mapsto 2])\\
&\quad \text{car eval}(1, [x \mapsto 2]) = 1\\
=\ &\ldots
\end{aligned}
$$

et ainsi de suite sans jamais pouvoir atteindre un résultat. La boucle infinie de ce programme déclenche du côté sémantique une fuite infinie dans les justifications. De ce fait, il est impossible d'obtenir un environnement ρ pour lequel on aurait exec(`x := 0;while (1) {x :=` $x \oplus 1$`}`, $\emptyset$) = ρ. Cet exemple met en évidence le fait que la fonction exec n'est pas une fonction totale : il existe des paires programme/environnement pour lesquelles elle n'est pas définie. C'est l'une des raisons pour lesquelles la sémantique est souvent définie comme une relation plutôt que comme une fonction (nous y reviendrons). Notez en outre que l'on peut observer le même phénomène dans le langage FUN en utilisant des fonctions récursives.

2.6 Approfondissement : subtilités sémantiques

Masquage. Bien que les variables y soient immuables, rien n'interdit en caml de redéfinir une nouvelle variable locale, du même nom qu'une variable existante.

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

Cette nouvelle version *masque* la précédente, et le `x` final de l'expression précédente sera donc associé à la valeur `2` donnée par la deuxième définition. Ce masquage ne vaut cependant que dans la partie de l'expression où la nouvelle variable existe. Ainsi dans le programme

```
let x = 1 in
let y = (let x = 2 in x) in
x + 2*y
```

le `x` de la dernière ligne vaut `1`, et `y` vaut `2`, soit la valeur de la deuxième définition de `x`. Notez que ce programme est à tous points de vue équivalent à la version suivante dans laquelle le deuxième `x` a été nommé `z` pour éviter les confusions.

```
let x = 1 in
let y = (let z = 2 in z) in
x + 2*y
```

De même, l'expression `let x = 1 in let x = 2 in x` de l'exemple précédent est équivalente à l'expression `let z = 1 in let x = 2 in x`.

Opérateurs paresseux. Nous avons donné ci-dessus une manière simple d'évaluer une expression de la forme e_1 `+` e_2 : d'abord évaluer les deux sous-expressions e_1 et e_2, puis additionner les résultats. Cette méthode ne s'étend cependant pas directement à tous les opérateurs binaires.

Comment calcule-t-on la valeur de l'expression suivante ?

```
n > 0  &&  x mod n == 0
```

Quelques scénarios.

- Si `n` vaut 3 et `x` vaut 6, les deux opérandes du `&&` s'évaluent à vrai, indiquant que 6 est bien un multiple de 3.
- Si `n` vaut 3 et `x` vaut 7, l'opérande de gauche s'évalue à vrai et celui de droite à faux, indiquant que 7 n'est pas un multiple de 3.
- Si en revanche `n` vaut 0 et `x` vaut 1, l'évaluation de l'opérande de droite conduirait à une erreur « division par zéro » et empêcherait le calcul d'aboutir.

En général, le troisième scénario ne se réalise pas, les langages réalisant la stratégie suivante, dite *paresseuse* :

1. évaluer l'opérande de gauche du `&&`,
2. puis en fonction du résultat,
   - si vrai, alors évaluer l'opérande de droite,
   - si faux, alors indiquer le résultat global faux sans évaluer l'opérande de droite.

Ainsi, les différents ordres possibles pour organiser l'interprétation d'un opérateur et l'évaluation de ses opérandes ne sont pas toujours équivalents. On appelle *strict* un opérateur dont tous les opérandes sont évalués systématiquement (comme `+`, `*` ou `<`), et *paresseux* un opérateur dont les opérandes ne sont évalués qu'en fonction des besoins (comme `&&` ou `||`).

Nous reviendrons sur ce phénomène et d'autres phénomènes associés lorsque nous manipulerons des fonctions.

Variables partagées (*aliasing*). En présence de variables mutables, la définition précise de l'effet d'une instruction d'affectation présente quelques subtilités, du fait d'une double signification des variables. Une variable est *un nom* qui, selon l'endroit où elle est utilisée, peut désigner des choses différentes :

- utilisé dans une expression arithmétique, un nom de variable désigne en général *la valeur* de cette variable ;
- utilisé à gauche du symbole d'affectation, un nom de variable désigne en général *l'emplacement mémoire* de cette variable (autrement dit *un pointeur*), dont on veut modifier le contenu.

Cette polysémie est partagée avec les autres formes d'expression que l'on peut trouver à gauche d'un symbole d'affectation, que l'on nomme collectivement les *valeurs gauches* : par exemple l'expression `t[i]` désignant une case d'un tableau ou l'expression `s.x` désignant un champ ou un attribut d'une structure ou d'un objet.

Dès lors, une question peut se poser à propos de la séquence d'instructions

```
x := 1;
y := x;
x := 2;
```

Qu'attendons-nous ici comme valeur finale pour `y` ? La réponse à cette question dépend du sens donné à l'affectation `y = x;`

— Il peut s'agir de l'affectation à `y` de la valeur courante de la variable `x` (en l'occurrence 1), après quoi les deux variables `x` et `y` restent indépendantes. Dans ce cas, `y` vaut toujours 1 à la fin.
— Il peut s'agir de définir `y` comme étant *la même variable* que `x`. On obtient alors deux noms désignant la même variable (on parle d'*aliasing*), et toute modification de cette variable commune vaut donc pour les deux noms. Dans cette situation, la valeur finale de `y` est 2.

Quizz : en C, en python, en java, laquelle de ces deux significations est retenue ? Dans chacun de ces langages, est-il possible d'obtenir l'autre effet ? Si oui, comment modifier l'instruction pour cela ?

Il est possible en C d'obtenir le pointeur vers l'emplacement mémoire d'une variable `x` avec l'expression `&x`, mais peu de langages actuels permettent un tel accès direct et illimité à ce pointeur. C++, java, python ou caml par exemple encapsulent les manipulations de pointeurs dans des constructions de plus haut niveau (objets, références, etc).

Pour définir une fonction exec en présence de pointeurs et d'aliasing, la notion d'environnement/de mémoire qui jusque là était simplement désignée par ρ doit être découpée en deux étages :

1. un environnement qui à chaque nom de variable associe l'emplacement mémoire (virtuel) associé,
2. une mémoire, qui à chaque emplacement mémoire virtuel associe la valeur qui y est stockée.

Ainsi, le phénomène d'*aliasing* se manifeste par l'existence dans l'environnement de deux variables associées au même emplacement mémoire, et qui partagent donc de fait une même valeur.

Nous reviendrons sur ce phénomène, et d'autres similaires, lorsque nous manipulerons des fonctions.

2.7 Approfondissement : preuve d'une propriété sémantique

Reprenons les expressions arithmétiques avec variables locales, et démontrons que la valeur d'une expression e ne dépend que des variables effectivement présentes dans e. Autrement dit, si deux environnements ρ et ρ' donnent les mêmes valeurs aux variables de e on a à coup sûr $\mathsf{eval}(e, \rho) = \mathsf{eval}(e, \rho')$.

Cet énoncé demande une petite clarification, car il y a deux manières différentes dont une variable `x` peut « apparaître » dans une expression e :

— la variable `x` peut être introduite par un `let`, à l'intérieur de e (on parle de variable ***liée***, ou locale),
— ou e peut faire référence à une variable `x` supposée définie par ailleurs (on parle de variable ***libre***).

La notion qui nous intéresse dans l'énoncé précédent est celle de variable libre. L'ensemble $\mathsf{fv}(e)$ des ***variables libres*** d'une expression e est défini par les équations

$$\begin{aligned}
\mathsf{fv}(\mathtt{n}) &= \emptyset \\
\mathsf{fv}(\mathtt{x}) &= \{x\} \\
\mathsf{fv}(e_1 \oplus e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(e_1 \otimes e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(\mathtt{let\ x = } e_1 \mathtt{\ in } e_2) &= \mathsf{fv}(e_1) \cup (\mathsf{fv}(e_2) \setminus \{x\})
\end{aligned}$$

On peut maintenant formaliser la propriété à démontrer :

Soient e une expression et ρ, ρ' deux environnements.
Si pour tout $x \in fv(e)$ on a $\rho(x) = \rho'(x)$,
alors $eval(e, \rho) = eval(e, \rho')$.

On démontre cette propriété par induction structurelle sur l'expression e. La propriété $P(e)$ qui nous intéresse est donc

« pour tous ρ, ρ', si $\forall x \in \mathsf{fv}(e), \rho(x) = \rho'(x)$, alors on a $\mathsf{eval}(e, \rho) = \mathsf{eval}(e, \rho')$ »

Le raisonnement par induction structurelle présente un cas pour chaque construction de la syntaxe abstraite.

— Cas n. Quelque soient ρ et ρ' on a $\mathsf{eval}(\mathsf{n}, \rho) = \mathsf{n} = \mathsf{eval}(\mathsf{n}, \rho')$.
— Cas x. Soient ρ, ρ' tels que $(\forall y \in \mathsf{fv}(\mathsf{x}), \rho(y) = \rho'(y))$, c'est-à-dire tels que $\rho(\mathsf{x}) = \rho'(\mathsf{x})$. On a $\mathsf{eval}(\mathsf{x}, \rho) = \rho(\mathsf{x}) = \rho'(\mathsf{x}) = \mathsf{eval}(\mathsf{x}, \rho')$.
— Cas Add. Soient e_1 et e_2 deux expressions telles que les propriétés $P(e_1)$ et $P(e_2)$ sont valides. Soient ρ, ρ' tels que $\forall x \in \mathsf{fv}(e_1 \oplus e_2), \rho(x) = \rho'(x)$. Comme $\mathsf{fv}(e_1 \oplus e_2) = \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)$, on a en particulier $\forall x \in \mathsf{fv}(e_1), \rho(x) = \rho'(x)$ et $\forall x \in \mathsf{fv}(e_2), \rho(x) = \rho'(x)$. Donc par $P(e_1)$ on a $\mathsf{eval}(e_1, \rho) = \mathsf{eval}(e_1, \rho')$ et par $P(e_2)$ on a $\mathsf{eval}(e_2, \rho) = \mathsf{eval}(e_2, \rho')$. Finalement,

$$\begin{array}{lll} & \mathsf{eval}(e_1 \oplus e_2, \rho) & \\ = & \mathsf{eval}(e_1, \rho) + \mathsf{eval}(e_2, \rho) & \text{par déf. de eval} \\ = & \mathsf{eval}(e_1, \rho') + \mathsf{eval}(e_2, \rho') & \text{par hyp. d'induction} \\ = & \mathsf{eval}(e_1 \oplus e_2, \rho') & \text{par déf. de eval} \end{array}$$

— Cas Mul similaire.
— Cas Let. Soient e_1 et e_2 deux expressions telles que les propriétés $P(e_1)$ et $P(e_2)$ sont valides. Soient ρ, ρ' tels que $\forall y \in \mathsf{fv}(\texttt{let x = } e_1 \texttt{ in } e_2), \rho(y) = \rho'(y)$. Par définition de eval on a

$$\mathsf{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho) = \mathsf{eval}(e_2, \rho[x \mapsto \mathsf{eval}(e_1, \rho)])$$

et

$$\mathsf{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho') = \mathsf{eval}(e_2, \rho'[x \mapsto \mathsf{eval}(e_1, \rho')])$$

On peut conclure par hypothèse d'induction $P(e_2)$, à condition de justifier que les deux environnements $\rho[x \mapsto \mathsf{eval}(e_1, \rho)]$ et $\rho'[x \mapsto \mathsf{eval}(e_1, \rho')]$ coïncindent sur $\mathsf{fv}(e_2)$. Soit $y \in \mathsf{fv}(e_2)$. On considère deux cas.
  — Si $y \neq x$, alors $y \in (\mathsf{fv}(e_2) \setminus \{x\})$ et donc $y \in \mathsf{fv}(\texttt{let x = } e_1 \texttt{ in } e_2)$. Alors par hypothèse $\rho(y) = \rho'(y)$ et on a donc

$$\begin{array}{lll} & (\rho[x \mapsto \mathsf{eval}(e_1, \rho)])(y) & \\ = & \rho(y) & y \neq x \\ = & \rho'(y) & \text{par hyp.} \\ = & (\rho'[x \mapsto \mathsf{eval}(e_1, \rho')])(y) & y \neq x \end{array}$$

  — Si $y = x$, alors

$$\begin{array}{lll} & (\rho[x \mapsto \mathsf{eval}(e_1, \rho)])(y) & \\ = & \mathsf{eval}(e_1, \rho) & y = x \\ = & \mathsf{eval}(e_1, \rho') & \text{par hyp. d'induction } P(e_1) \\ = & (\rho'[x \mapsto \mathsf{eval}(e_1, \rho')])(y) & y = x \end{array}$$

Les deux environnements donnent donc toujours la même valeur pour $y \in \mathsf{fv}(e_2)$, donc par hypothèse d'induction $P(e_2)$ on conclut que $\mathsf{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho) = \mathsf{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho')$.

Exercice 2.2. Dans un langage d'expressions arithmétiques où les variables locales sont immuables, l'expression $\texttt{let x = } e_1 \texttt{ in } e_2$ est équivalente à l'expression e_2 dans laquelle chaque occurrence de x aurait été remplacée par l'expression e_1. Par ***équivalente*** on signifie que les deux expressions produisent le même résultat, quelque soit le contexte ρ dans lequel on les évalue toutes deux.

Définition de la ***substitution*** $e[x := s]$ de chaque occurrence de x dans e par s.

$$\begin{array}{rcl} \mathsf{n}[x := s] & = & \mathsf{n} \\ \mathsf{y}[x := s] & = & \begin{cases} s & \text{si } x = y \\ \mathsf{y} & \text{sinon} \end{cases} \\ (e_1 \oplus e_2)[x := s] & = & (e_1[x := s]) \oplus (e_2[x := s]) \\ (e_1 \otimes e_2)[x := s] & = & (e_1[x := s]) \otimes (e_2[x := s]) \\ (\texttt{let y = } e_1 \texttt{ in } e_2)[x := s] & = & \begin{cases} \texttt{let y = } (e_1[x := s]) \texttt{ in } e_2 & \text{si } x = y \\ \texttt{let y = } (e_1[x := s]) \texttt{ in } (e_2[x := s]) & \text{si } x \neq y \text{ et } y \notin \mathsf{fv}(s) \end{cases} \end{array}$$

Pour justifier l'équivalence entre la construction Let et la substitution, démontrer la propriété

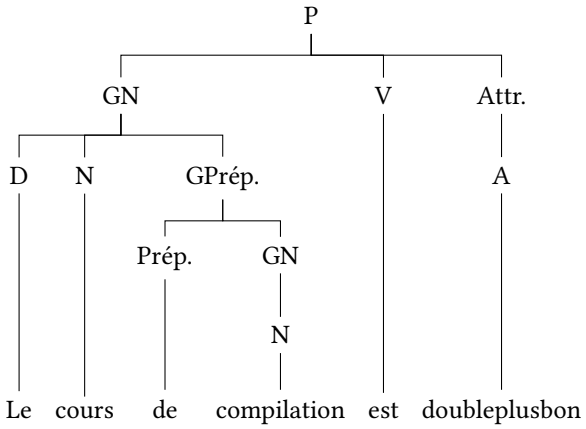
$$\forall x, e_1, e_2, \rho,\ x \notin \mathsf{fv}(e_1) \ \Rightarrow\ \mathsf{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, \rho) \ =\ \mathsf{eval}(e_2[x := e_1], \rho)$$

par induction sur la struture de e_2.

3 Analyse syntaxique d'un langage impératif

On reprend le langage IMP du chapitre précédent, en s'intéressant cette fois à l'analyse de son code source, c'est-à-dire du fichier texte contenant un programme, afin d'en extraire un arbre de syntaxe abstraite. On part donc d'une simple séquence de caractères, que l'on veut d'abord découper en mots (analyse lexicale), avant de regrouper ces mots en un ensemble de phrases structurées (analyse grammaticale).

Un concept central du chapitre est la notion de ***grammaire***, qui décrit la manière dont différents éléments peuvent s'agencer pour former une phrase bien structurée.



3.1 Le problème à l'envers : affichage

Avant de décrire l'analyse syntaxique elle-même, c'est-à-dire le passage d'une séquence de lexèmes à un arbre de syntaxe, nous allons regarder le problème inverse : partant d'un arbre de syntaxe, nous allons chercher à afficher joliment son contenu.

Prenons un noyaux simpliste d'expressions arithmétiques, avec constantes entières, additions et multiplications.

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

On peut ainsi considérer la valeur caml `e`

```
let e = Add(Cst 1, Mul(Add(Cst 2, Cst 3), Cst 4))
```

représentant l'expression `1+(2+3)*4`.

Résumons les règles d'écriture des expressions arithmétiques basées sur ces opérateurs :

- une constante entière est une expression arithmétique,
- la concaténation e_1 `+` e_2 d'une première expression arithmétique e_1, du symbole `+` et d'une deuxième expression arithmétique e_2 forme une expression arithmétique,
- la concaténation e_1 `*` e_2 d'une première expression arithmétique e_1, du symbole `*` et d'une deuxième expression arithmétique e_2 forme une expression arithmétique,
- la concaténation `(` e `)` d'une parenthèse ouvrante `(`, d'une expression arithmétique e et d'une parenthèse fermante `)` forme une expression arithmétique.

En notant E l'ensemble des expressions arithmétiques, on peut résumer ces règles avec la notation abrégée suivante :

$$
\begin{array}{rcl}
E & ::= & n \\
 & | & E \texttt{ + } E \\
 & | & E \texttt{ * } E \\
 & | & \texttt{(} E \texttt{)}
\end{array}
$$

Partant d'une valeur caml telle que `e` on veut donc afficher l'expression correspondante, ou disons renvoyer une chaîne de caractères représentant cette expression, en respectant les règles d'écriture que nous venons d'énoncer.

Une version naïve d'une telle fonction pourrait s'écrire ainsi.

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> pp e1 ^ " + " ^ pp e2
  | Mul(e1, e2) -> pp e1 ^ " * " ^ pp e2
```

Chaque opération binaire est directement traduite en la combinaison de ses deux opérandes par l'opérateur correspondant. Or, une telle fonction ne produit pas le résultat escompté.

```
# pp e
- : string = "1 + 2 + 3 * 4"
```

Il manque des parenthèses pour que cette chaîne décrive bien la structure de l'expression d'origine.

Modifier notre fonction pour ajouter des parenthèses partout n'est guère satisfaisant non plus, puisque cela écrit des parenthèses superflues, que l'on ne souhaite pas voir dans un affichage « joli ».

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> "(" ^ pp e1 ^ " + " ^ pp e2 ^ ")"
  | Mul(e1, e2) -> "(" ^ pp e1 ^ " * " ^ pp e2 ^ ")"
```

```
# pp e
- : string = "(1 + ((2 + 3) * 4))"
```

En décidant de ne jamais mettre de parenthèses autour des multiplications, ces parenthèses superflues peuvent être limitées, mais pas totalement éliminées.

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> "(" ^ pp e1 ^ " + " ^ pp e2 ^ ")"
  | Mul(e1, e2) -> pp e1 ^ " * " ^ pp e2
```

```
# pp e
- : string = "(1 + (2 + 3) * 4)"
```

Il faut donc encore regarder d'un peu plus près la structure de nos expressions arithmétiques.

On peut améliorer notre affichage en distinguant plusieurs catégories de positions à l'intérieur d'une expression arithmétique, qui doivent être traitées différemment par l'afficheur. En l'occurrence, l'écriture des opérandes d'une multiplication va parfois nécessiter des parenthèses qui auraient été inutiles pour la même expression placée à un autre endroit. Pour rendre compte de cela on peut séparer notre grammaire en deux catégories : E pour les expressions ordinaires et M pour les opérandes d'une multiplication. La différence entre les deux est qu'une opération d'addition doit être entourée de parenthèses lorsqu'elle est l'opérande d'une opération de multiplication.

$$
\begin{array}{rcl}
E & ::= & n \\
 & | & E + E \\
 & | & M * M \\
M & ::= & n \\
 & | & M * M \\
 & | & (E + E)
\end{array}
$$

On peut maintenant créer une nouvelle fonction d'affichage, plus fine, donnée en réalité par deux fonctions mutuellement récursives, chacune correspondant à l'une des deux positions E ou M que nous venons d'identifier.

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> pp e1 ^ " + " ^ pp e2
  | Mul(e1, e2) -> pp_m e1 ^ " * " ^ pp_m e2
and pp_m = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> "(" ^ pp e1 ^ " + " ^ pp e2 ^ ")"
  | Mul(e1, e2) -> pp_m e1 ^ " * " ^ pp_m e2
```

On obtient cette fois un affichage de notre expression exemple qui fait apparaître exactement les parenthèses nécessaires, et pas une de plus.

```
# pp e
- : string = "1 + (2 + 3) * 4"
```

On peut cependant améliorer encore une fois notre analyse et la fonction d'affichage qui s'en déduit : pour l'instant, les mêmes motifs apparaissent plusieurs fois dans notre description, et donc également plusieurs fois dans le code. On peut factoriser la description des expressions arithmétiques de manière à ce que les éléments n, E `+` E et M `*` M n'apparaissent plus qu'une fois chacun. Pour cela, on coupe maintenant en trois catégories : les expressions ordinaires E, les expressions multiplicatives M et les expressions atomiques A.

$$
\begin{array}{rcl}
E & ::= & E \texttt{ + } E \\
 & | & M \\
M & ::= & M \texttt{ * } M \\
 & | & A \\
A & ::= & n \\
 & | & \texttt{(} E \texttt{)}
\end{array}
$$

La fonction d'affichage peut encore être adaptée à cette nouvelle interprétation de la structure des expressions arithmétiques, pour donner un code équivalent au précédent mais sans plus aucune redondance.

```
let rec pp = function
  | Add(e1, e2) -> pp e1 ^ " + " ^ pp e2
  | e -> pp_m e
and pp_m = function
  | Mul(e1, e2) -> pp_m e1 ^ " * " ^ pp_m e2
  | e -> pp_a e
and pp_a = function
  | Cst n -> string_of_int n
  | e -> "(" ^ pp e ^ ")"
```

3.2 Grammaires et dérivations

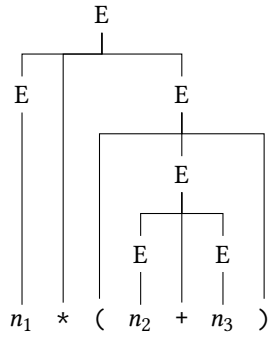
La ***grammaire*** d'un langage décrit les structures de phrases légales dans ce langage. On la définit à l'aide de deux éléments en plus du lexique :

— les différentes catégories de (fragments de) phrases du langage,
— les manières dont différents éléments peuvent être regroupés pour former un fragment de l'une ou l'autre des catégories (données par des règles de combinaisons).

Les phrases bien formées sont alors celles dont les mots peuvent être regroupés d'une manière compatible avec les règles de combinaison.

Structure grammaticale d'une phrase. On peut exposer la structure grammaticale d'une phrase en dessinant un arbre dont les feuilles sont les mots de la phrase (dans l'ordre), et dont chaque nœud interne dénote le regroupement de ses fils. Avec la première grammaire décrite pour les expressions arithmétiques nous pouvons construire ainsi la structure de la phrase n_1`*(`n_2`+`n_3`)`.

$$
\begin{array}{rcl}
E & ::= & n \\
 & | & E \texttt{ + } E \\
 & | & E \texttt{ * } E \\
 & | & \texttt{(} E \texttt{)}
\end{array}
$$



Grammaire algrébrique. Formellement, on définit une ***grammaire algébrique*** par un quadruplet (T, N, S, R) où :

- T est un ensemble de symboles ***terminaux***, désignant des mots,
- N est un ensemble de symboles ***non terminaux***, désignant des groupes de mots,
- S est le ***symbole de départ***, qui décrit une phrase complète,
- R est un ensemble de ***règles de production***, c'est-à-dire de paires $X \prec a_1 \dots a_k$ associant un symbole non terminal X et une séquence $a_1 \dots a_k$ de symboles (terminaux ou non) formant un groupe de type X correct.

Les symboles terminaux sont les *lexèmes* produits par l'analyse lexicale, et les symboles non terminaux représentent les *catégories* évoquées plus haut. Le symbole de départ S est un symbole non terminal. Note : il n'est pas interdit d'avoir une règle de production associant un symbole non terminal X à une séquence vide (c'est-à-dire une séquence de zéro symboles). On la note alors $X \prec \varepsilon$.

Une grammaire simple pour les expressions arithmétiques. Dans une grammaire naïve des expressions arithmétiques, on peut prendre :

- les symboles terminaux `+`, `*`, `(`, `)` ainsi qu'un terminal `n` pour une constante entière,
- le symbole non terminal E,
- le symbole de départ E,
- les règles de production $E \prec$ `n`, $E \prec E$`+`E, $E \prec E$`*`E et $E \prec$ `(`E`)`, encore écrites

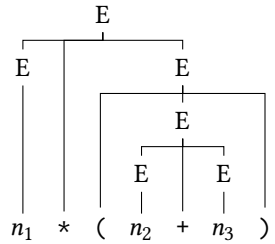
$$\begin{array}{lcl} E & ::= & \texttt{n} \\ & | & E\ \texttt{+}\ E \\ & | & E\ \texttt{*}\ E \\ & | & \texttt{(}\ E\ \texttt{)} \end{array}$$

Dérivation. Étant donnée une grammaire (T, N, S, R), un ***arbre de dérivation*** est un arbre vérifiant les conditions suivantes :

- chaque nœud interne est étiqueté par un symbole de N,
- chaque feuille est étiquetée par un symbole de T ou de N, ou par le symbole ε,
- pour chaque nœud interne étiqueté par un symbole $X \in N$, les étiquettes de ses fils prises dans l'ordre forment une séquence β telle que $X \prec \beta \in R$. Exception : lorsque $X \prec \varepsilon \in R$, on peut aussi avoir un nœud interne étiqueté par X avec un unique fils étiqueté par la séquence vide ε.

Techniquement, ce symbole ε n'est ni dans T ni dans N. Il désigne une *absence* de symbole.

On dit qu'une phrase m est ***dérivable*** à partir d'un symbole non terminal X s'il existe un arbre de dérivation dont la racine est étiquetée par X et dont les feuilles, prises dans l'ordre, forment cette phrase. Les phrases ***dérivables*** de la grammaire sont celles qui sont dérivables à partir du symbole de départ S. L'arbre



est un arbre de dérivation pour l'expression arithmétique n_1`*(`n_2`+`n_3`)` suivant la grammaire naïve, à partir de son symbole de départ E.

Une grammaire plus précise pour les expressions arithmétiques. La description plus fine des expressions arithmétiques que nous avons vue en fin de section précédente donne elle une grammaire distinguant trois catégories de fragments notées E, M et A et précisant les règles suivantes :

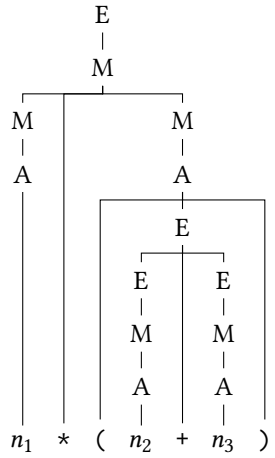
- le regroupement d'un fragment E, du mot `+` et d'un deuxième fragment E forme un fragment E,
- un fragment M seul forme également un fragment E,
- le regroupement d'un fragment M, du mot `*` et d'un deuxième fragment M forme un fragment M,
- un fragment A seul forme également un fragment M,

— un mot n seul forme un fragment A,
— le regroupement du mot `(`, d'un fragment E et du mot `)` forme un fragment A.

Autrement dit, nous avons une nouvelle grammaire définie par :
— les mêmes symboles terminaux `+`, `*`, `(`, `)` et, pour chaque constante entière, `n`,
— les symboles non terminaux E, M et A,
— le symbole de départ E,
— les règles de production $E \prec E$`+`E, $E \prec M$, $M \prec M$`*`M, $M \prec A$, $A \prec$ `n` et $A \prec$ `(`E`)`, encore écrites

$$\begin{array}{rcl} E & ::= & E \texttt{ + } E \\ & | & M \\ M & ::= & M \texttt{ * } M \\ & | & A \\ A & ::= & n \\ & | & \texttt{(} E \texttt{)} \end{array}$$

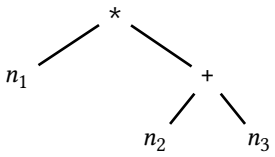
Nous avons pour n_1`*(`n_2`+`n_3`)` un nouvel arbre de dérivation à partir du symbole de départ E.



Note : les arbres de dérivation matérialisent la structure d'une phrase, exprimée en fonction des règles de la grammaire. Il s'agit d'un concept différent de celui d'arbre de syntaxe abstraite. En l'occurrence, l'arbre de syntaxe abstraite associé à l'expression `n1 * (n2 + n3)` s'écrirait en caml

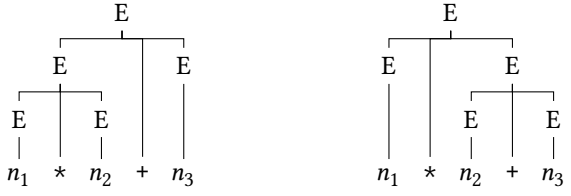
```
Mul(Cst n1, Add(Cst n2, Cst n3))
```

et serait dessiné ainsi, indépendamment de la grammaire utilisée pour caractériser la structure des expressions bien formées.



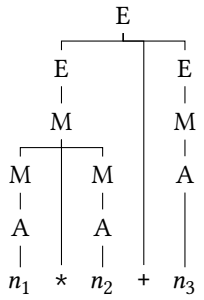
L'objectif d'un analyseur syntaxique est de construire cet arbre de syntaxe abstraite. L'arbre de dérivation, lui, décrit les différentes étapes de l'analyse réalisée, mais n'est pas construit explicitement par l'analyseur.

Ambiguïtés Une grammaire est ***ambiguë*** s'il existe une phrase qui peut être dérivée par deux arbres différents. Si l'on considère par exemple la phrase n_1`*`n_2`+`n_3 et qu'on l'analyse en suivant les règles de la grammaire naïve, deux arbres de dérivation sont possibles.



Le premier identifie comme un groupe la sous-expression $n_1 * n_2$, et correspond bien à l'interprétation conventionnelle des expressions arithmétiques. Le deuxième en revanche forme un groupe avec le fragment $n_2 + n_3$ et garde la multiplication à part. Cette deuxième interprétation est incorrecte par rapport aux conventions usuelles.

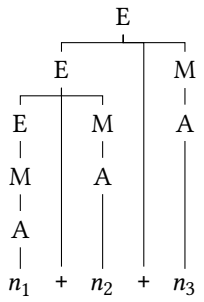
La deuxième grammaire ne présente pas d'ambiguïté sur cette expression, qui ne peut être dérivée que par l'arbre suivant.



On peut vérifier en revanche que cette grammaire plus élaborée reste ambiguë, par exemple avec la phrase $n_1 + n_2 + n_3$. On peut obtenir une grammaire non ambiguë avec (par exemple) la variante minime suivante.

$$
\begin{array}{lcl}
E & ::= & E \ + \ M \\
 & | & M \\
M & ::= & M \ * \ A \\
 & | & A \\
A & ::= & n \\
 & | & (\ E\ )
\end{array}
$$

On a alors notamment un unique arbre de dérivation pour la phrase $n_1 + n_2 + n_3$.



3.3 Analyse récursive descendante

L'analyse syntaxique prend en entrée une séquence de lexèmes, et doit vérifier que cette séquence est cohérente avec les règles de grammaire du langage. Pour chaque entrée à laquelle on applique un analyseur syntaxique, on attend l'une des issues suivantes :

- en cas de succès, un arbre de syntaxe abstraite donnant la structure de la séquence lue en entrée,
- en cas d'échec, la localisation et l'explication de ce qui est grammaticalement incohérent dans l'entrée.

Dans cette section, nous ne construirons pas encore l'arbre de syntaxe abstraite et nous contenterons d'indiquer si l'analyse a réussi (la phrase a une structure correcte) ou a échoué (la phrase a une structure incorrecte).

Pour un minimum d'efficacité algorithmique, on va éviter les algorithmes coûteux consistant à regarder toutes les décompositions possibles d'un phrase et préférer une lecture de gauche à droite, sans retour en arrière.

La technique la plus simple à programmer revient à construire un arbre de dérivation en partant de la racine, c'est-à-dire du symbole de départ, et en expansant les symboles non terminaux en essayant de former la phrase cible. On parle d'analyse *descendante*, ou *top-down*, puisque l'on découvre l'arbre de dérivation de haut en bas.

On réalise simplement un tel analyseur en définissant, pour chaque symbole non terminal X, une fonction f_X prenant en entrée une séquence de lexèmes (c'est-à-dire une séquence de

symboles terminaux), essayant de reconnaître dans un préfixe de cette séquence un fragment de phrase correspondant à la structure X, et renvoyant les lexèmes qui n'ont pas été utilisés (ou échouant si elle ne parvient pas à reconnaître un fragment correspondant à X).

Construction manuelle d'un analyseur. Prenons notre grammaire non ambiguë des expressions arithmétiques.

$$\begin{array}{rcl} E & ::= & E \texttt{ + } M \\ & | & M \\ M & ::= & M \texttt{ * } A \\ & | & A \\ A & ::= & n \\ & | & \texttt{(} E \texttt{)} \end{array}$$

Définissons ensuite un type de données pour les lexèmes (i.e. symboles terminaux),

```
type token =
  | NUM of int
  | PLUS
  | STAR
  | LPAR
  | RPAR
```

et voyons à quoi pourrait ressembler une fonction d'analyse pour le symbole non terminal A. Cette fonction prend en paramètre une liste de lexèmes à analyser, et renvoie la liste des lexèmes qui n'ont pas servi.

```
let validate_atom: token list -> token list = function
```

La grammaire donne plusieurs règles d'expansion pour A. Pour éviter une explosion de l'analyse, on ne s'autorisera pas une approche par essais et erreurs (essayer d'abord avec une règle, puis avec l'autre si cela n'a pas fonctionné, et ainsi de suite). Il faut donc choisir, en fonction du contexte, la règle qui aura les meilleures chances de mener à une analyse réussie. Les informations à notre disposition pour choisir sont :

- notre connaissance de la grammaire,
- le prochain lexème de l'entrée.

En l'occurrence ici, le premier lexème de l'entrée permet sans ambiguïté de sélectionner une unique règle.

- Si le prochain lexème est une constante `n`, alors l'analyse va réussir avec la seule application de la règle $A \prec \texttt{n}$ et il suffit de renvoyer la suite de la liste.

  ```
      | NUM _ :: l -> l
  ```

- Si le prochain lexème est une parenthèse ouvrante `(`, alors l'analyse *peut* réussir avec la règle $A \prec \texttt{(}E\texttt{)}$, si la suite de la liste correspond bien à la séquence restante E`)`. Il faut donc d'abord chercher à reconnaître E dans la suite de la liste, ce que l'on fait avec un appel à une fonction `validate_expr` dédiée à la reconnaissance de cette structure (toutes les fonctions de reconnaissance vont donc être mutuellement récursives), puis vérifier que la séquence continue avec une parenthèse fermante.

  ```
      | LPAR :: l -> let l' = validate_expr l in
                     let l'' = match l' with
                       | RPAR :: l'' -> l''
                       | _ -> failwith "syntax error"
                     in
                     l''
  ```

- Dans tous les autres cas, aucune règle ne peut s'appliquer : la liste de lexèmes ne contient pas de préfixe correspondant à la structure A, et l'analyse doit échouer.

  ```
      | _ -> failwith "syntax error"
  ```

Ne reste donc pour compléter l'analyseur qu'à définir des fonctions analogues pour les autres symboles non terminaux. Le cas du symbole E n'est cependant pas si simple : nous avons deux expansions possibles $E \prec E$`+`M et $E \prec M$, entre lesquelles on ne peut choisir à coup sûr

en ne connaissant que le prochain symbole de l'entrée (ni même en s'autorisant à regarder un certain nombre $k \geqslant 1$ de lexèmes, puisque le symbole + distinguant la première expansion peut apparaître arbitrairement loin).

Pour compléter notre analyseur, il va falloir transformer la grammaire de sorte à simplifier le choix entre les différentes règles à expanser. On peut par exemple modifier les règles du symbole E en les suivantes

$$\begin{array}{rcl} E & ::= & M \texttt{ + } E \\ & | & M \end{array}$$

Transformer ainsi une grammaire sans modifier l'ensemble des phrases valides est un exercice délicat. On verra plus tard des outils qui permettent d'éviter d'avoir à le faire trop souvent.

de sorte à faire ressortir un préfixe commun aux deux règles : pour reconnaître un fragment E on commence dans tous les cas par chercher à reconnaître un fragment M, puis on peut ou non avoir une suite introduite par +.

```
and validate_expr l =
  let l' = validate_m_expr l in
  let l'' = match l' with
    | PLUS :: l'' -> validate_expr l''
    | _ -> l'
  in
  l''
```

On peut appliquer une modification similaire aux règles du symbole non terminal M pour écrire la fonction `validate_m_expr` et compléter l'analyseur. La fonction principale de reconnaissance d'une expression complète consiste à alors à appliquer `validate_expr` à la liste de lexèmes complète et vérifier que l'on est allé jusqu'au bout de la séquence, c'est-à-dire qu'il ne reste plus qu'une liste vide.

```
let validate l =
  validate_expr l = []
```

Notez que les fonctions qui viennent d'être décrites peuvent être factorisées pour obtenir le code compact et complet suivant.

```
let expect t = function
  | t' :: l when t = t' -> l
  | _ -> failwith "syntax error"

let rec validate_e l = validate_m l |> validate_e'
and validate_e' = function
  | PLUS :: l -> validate_e l
  | l -> l
and validate_m l = validate_a l |> validate_m'
and validate_m' = function
  | STAR :: l -> validate_m' l
  | l -> l
and validate_a = function
  | LPAR :: l -> validate_e l |> expect RPAR
  | NUM _ :: l -> l
  | _ -> failwith "syntax error"

let validate l = validate_e l = []
```

Cette factorisation du code est d'ailleurs liée à la factorisation possible suivante de la grammaire.

$$\begin{array}{rcl} E & ::= & M\,E' \\ E' & ::= & \texttt{+ } E \\ & | & \varepsilon \\ M & ::= & A\,M' \\ M' & ::= & \texttt{* } M \\ & | & \varepsilon \\ A & ::= & n \\ & | & \texttt{(} E \texttt{)} \end{array}$$

Enfin, on peut imaginer une variante de ce programme d'analyse qui construit au passage l'arbre de syntaxe abstraite de l'expression reconnue, plutôt que de simplement déclarer que l'expression lue est bien formée. Les fonctions `parse_*` renvoient donc une expression en plus de la liste des mots restant à analyser.

Cette version a déjà été aperçue au chapitre 1.

```
let rec parse_expr l = parse_m_expr l |> expand_sum
and expand_sum = function
  | e1, PLUS :: l -> let e2, l' = parse_m_expr l in
                     expand_sum (Add(e1, e2), l')
  | r -> r
and parse_m_expr l = parse_atom l |> expand_prod
and expand_prod = function
  | e1, STAR :: l -> let e2, l' = parse_atom l in
                     expand_prod (Mul(e1, e2), l')
  | r -> r
and parse_atom = function
  | NUM n :: l -> Cst n, l
  | LPAR :: l -> (match parse_expr l with
                  | e, RPAR :: l' -> e, l'
                  | _ -> failwith "syntax error")
  | _ -> failwith "syntax error"

let parse (input: token list): expr =
  match parse_expr input with
  | e, [] -> e
  | _ -> failwith "syntax error"
```

Note : si vous regardez de près les fonctions `parse_*` *de ce dernier programme, vous pourrez observer qu'elles apportent encore une légère variation par rapport à la dernière version de la grammaire des expressions arithmétiques. Quelle est cette nouvelle grammaire ? Quel programme aurait-on obtenu en suivant directement la version précédente de la grammaire ? En quoi diffèrent-ils ?*

3.4 Approfondissement : un analyseur complet pour IMP

Appliquons ces techniques d'analyse au langage IMP. On se donne une grammaire avec les symboles terminaux suivants :

- un symbole *n* pour chaque constante entière,
- un symbole *x* pour chaque nom de variable,
- les symboles `+`, `*`, `<`, `&&`, `(`, `)`, `{`, `}`, `:=`, `;`,
- les mots-clés `print`, `while`, `if`, `else`.

Un programme complet (symbole de départ *P*) est simplement donné par une séquence d'instructions. Une séquence d'instruction (symbole non terminal *S*) peut être vide, et est sinon formée d'une première instruction, suivie d'une séquence. Une instruction (symbole non terminal *I*) est soit une affectation, soit une boucle, soit un branchement, soit un affichage. Une expression (symbole non terminal *E*) est soit une expression atomique, soit une opération binaire. Une expression atomique (symbole non terminal *A*) est soit une constante entière, soit une variable, soit une expression entre parenthèses.

$$
\begin{array}{lcll}
P & ::= & S & \\
S & ::= & \varepsilon & \\
 & | & I\,S & \\
I & ::= & x := E\ ; & \\
 & | & \texttt{while}\ (\ E\ )\ \{\ S\ \} & \\
 & | & \texttt{if}\ (\ E\ )\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \} & \\
 & | & \texttt{print}\ (\ E\ )\ ; & \\
E & | & A & \\
 & | & E\ o\ E & \text{avec } o \in \{\texttt{+}, \texttt{*}, \texttt{<}, \texttt{\&\&}\} \\
A & ::= & n & \\
 & | & x & \\
 & | & (\ E\ ) &
\end{array}
$$

La grammaire ne le montre pas, mais on s'assurera dans le code de l'analyseur que les priorité usuelles des différents opérateurs binaires sont bien respectées.

Analyse lexicale. On définit un type `token` pour les mots. Il contient précisément les symboles terminaux énumérés dans la grammaire, plus un élément spécial `EOI` matérialisant la fin de l'entrée.

```
type token =
  | NUM of int                    (*  n             *)
  | VAR of string                 (*  x             *)
  | PLUS | STAR | LT | AND        (*  +  *  <  &&   *)
  | LPAR | RPAR | BEGIN | END     (*  (  )  {  }    *)
  | SET | SEMI                    (*  :=  ;         *)
  | PRINT | WHILE | IF | ELSE
  | EOI
```

On définit une fonction `tokenizer`, qui prend en paramètre l'entrée sous la forme d'une chaîne de caractères et qui produit un analyseur lexical, c'est-à-dire une fonction `next_token` qui, à chaque appel, renvoie le lexème suivant de l'entrée. Cette fonction utilise une référence interne `i` mémorisant la position actuelle dans l'entrée. À chaque fois qu'un lexème est produit, cette référence est mise à jour.

Exercice : ajouter les quelques opérateurs binaires qui manquent pour traiter le programme exemple d'exponentiation rapide.

```
let tokenizer (s: string): unit -> token =
  let i = ref 0 in
  let rec next_token () = match s.[!i] with
    | '+' -> incr i; PLUS
    | '*' -> incr i; STAR
    | '<' -> incr i; LT
    | '(' -> incr i; LPAR
    | ')' -> incr i; RPAR
    | '{' -> incr i; BEGIN
    | '}' -> incr i; END
    | ';' -> incr i; SEMI
    | ':' -> if s.[!i+1] = '=' then (i := !i+2; SET)
             else failwith "lexical error"
    | '&' -> if s.[!i+1] = '&' then (i := !i+2; AND)
             else failwith "lexical error"
```

Lorsque le prochain caractère est un chiffre on appelle une fonction auxiliaire pour détecter une séquence de chiffres complète, qu'on décode en un unique nombre. Lorsque le prochain caractère est alphabétique, on procède de même pour récupérer une séquence de lettres, qui soit est un mot-clé, soit est considéré comme étant un identifiant de variable.

```
    | '0'..'9' -> let k = scan_int 1 in
                  let t = String.sub s !i k in
                  i := !i + k;
                  NUM (int_of_string t)
    | 'a'..'z' -> let k = scan_word 1 in
                  let t = String.sub s !i k in
                  i := !i + k;
                  (match t with
                   | "print" -> PRINT
                   | "while" -> WHILE
                   | "if"    -> IF
                   | "else"  -> ELSE
                   | _       -> VAR t)
```

On ignore les espaces et les retours à la ligne en relançant immédiatement l'analyseur sur les caractères suivants. Tous les caractères non mentionnés jusqu'ici sont considérés comme des erreurs. Enfin, si la fin de l'entrée est atteinte, on rattrape l'erreur déclenchée par l'accès invalide `s.[!i]` et on renvoie la valeur spéciale `EOI`.

```
    | ' ' | '\n' -> incr i; next_token ()
    | c -> failwith (Printf.sprintf "unknown character %c" c)
    | exception e -> EOI
```

Les fonctions auxiliaires nécessaires au décodage des entiers et des identifiants se contentent de parcourir l'entrée jusqu'à rencontrer un caractère qui est autre chose qu'un nombre ou une lettre, ou jusqu'à atteindre la fin de l'entrée, et renvoient la longueur du segment parcouru.

```
  and scan_int k = match s.[!i+k] with
    | '0'..'9' -> scan_int (k+1)
    | _ -> k
    | exception e -> k
  and scan_word k = match s.[!i+k] with
    | 'a'..'z' -> scan_word (k+1)
    | _ -> k
    | exception e -> k
```

Enfin, le résultat de `tokenizer` est la fonction `next_token` elle-même.

```
  in
  next_token
```

Syntaxe abstraite. On se donne la définition suivante pour les arbres de syntaxe abstraite que doit produire notre analyse syntaxique.

```
type prog = instr list
and instr =
  | Print of expr
  | Set   of string * expr
  | While of expr * instr list
  | If    of expr * instr list * instr list
and expr =
  | Cst of int
  | Var of string
  | Bop of bop * expr * expr
and bop = Add | Mul | Lt | And
```

Pour traduire les différentes priorités entre opérateurs, on affecte à chaque opérateur binaire un *niveau de priorité* entier. Plus ce niveau est élevé, plus l'opérateur est prioritaire. Ainsi par exemple : le niveau de priorité de la multiplication est supérieur à celui de l'addition.

```
let prio = function
  | AND  -> 1
  | LT   -> 2
  | PLUS -> 3
  | STAR -> 4
  | _    -> failwith "prio: expected operator"
let root_prio = 0
```

Analyse syntaxique. La fonction principale d'analyse crée un analyseur lexical à l'aide de la fonction `tokenizer` vue plus haut et définit une référence `current_token` enregistrant le lexème courant. On se donne également deux auxiliaires : une fonction `next` qui met à jour le lexème courant, et une fonction `take_token` qui consulte le lexème courant avant de le mettre à jour.

```
let parse (s: string): prog =
  let next_token = tokenizer s in
  let current_token = ref (next_token()) in
  let next() = current_token := next_token() in
  let take_token() = let t = !current_token in next(); t in
```

On aura fréquemment besoin de vérifier que le prochain lexème est d'un type particulier, par exemple une parenthèse ou un point-virgule. On prévoit pour cela une fonction auxiliaire `expect`, qui vérifie et consomme le prochain lexème. On peut faire produire à cette fonction des messages d'erreur relativement précis, pour peu de se donner une fonction `string_of_token` qui retraduit nos lexèmes en chaînes de caractères.

```
  let expect s =
    let t = take_token() in
    if t <> s then
      failwith ("syntax error, expected " ^ (string_of_token s)
                ^ " but got " ^ (string_of_token t))
  in
```

On réalise ensuite un analyseur récursif descendant, c'est-à-dire avec une fonction d'analyse pour chaque symbole non terminal. Pour analyser une séquence d'instructions on commence par tester les deux cas possibles de fin d'une séquence : fin de l'entrée, ou accolade fermante. Sinon, on décode une première instruction, puis à nouveau une séquence (qui elle-même peut être vide).

```
let rec parse_instrs () = match !current_token with
  | EOI | END -> []
  | _ -> let i = parse_instr () in
         let is = parse_instrs () in
         i :: is
```

Pour analyser une instruction seule, on regarde le premier symbole, qui suffit à déterminer la forme de l'instruction. Ainsi, si le premier symbole est une variable, on cherche à reconnaître un motif de la forme x `:=` E `;`, tandis que si la premier symbole est le mot-clé **`while`**, on cherche à reconnaître un motif de la forme `while (` E `) {` S `}`. En revanche, si le premier symbole est autre chose qu'une variable ou qu'un mot-clé `print`, **`while`** ou **`if`**, alors l'entrée ne peut pas être une instruction bien formée, et on échoue. Dans ce code, on factorise également la reconnaissance des motifs récurrents `(` E `)` et `{` S `}` à l'aide de deux fonctions auxiliaires `parse_cond` et `parse_block`.

```
and parse_instr () = match take_token() with
  | VAR x -> expect SET;
             let e = parse_expr root_prio in
             expect SEMI;
             Set(x, e)
  | PRINT -> let e = parse_cond () in
             expect SEMI;
             Print(e)
  | WHILE -> let e = parse_cond () in
             let is = parse_block () in
             While(e, is)
  | IF    -> let e = parse_cond () in
             let is1 = parse_block () in
             expect ELSE;
             let is2 = parse_block () in
             If(e, is1, is2)
  | _ -> failwith "syntax error"

and parse_cond () =
  expect LPAR;
  let e = parse_expr root_prio in
  expect RPAR;
  e

and parse_block () =
  expect BEGIN;
  let is = parse_instrs () in
  expect END;
  is
```

À la section précédente, nous avons géré les priorités entre opérateurs dans les expressions arithmétiques en distinguant deux paires de fonctions `parse_expr`/`expand_sum` et `parse_m_expr`/`expand_prod`, correspondant aux deux niveaux de priorité de l'addition et de la multiplication. Ici, les comparaisons et les fonctions booléennes apportent de nouveaux niveaux, que l'on pourrait traduire par de nouvelles fonctions.

Au lieu de cela, nous allons utiliser seulement deux fonctions `parse_expr` et `expand_op`, *paramétrées par un niveau de priorité* `p`, qui généralisent les fonctions précédentes.

La fonction `parse_expr` analyse d'abord la première expression atomique, puis appelle `expand_op` avec le même niveau de priorité. La fonction `expand_op` compare alors la priorité du prochain opérateur avec la priorité `p` courante. L'opération suivante n'est consommée que si l'opérateur a un niveau strictement supérieur au niveau courant. Lorsque que l'on analyse le deuxième opérande, à nouveau avec `parse_expr`, on utilise la priorité du dernier opérateur rencontré, nécessairement strictement supérieure à celle d'origine. Ainsi, on ne regroupera dans `e2` que des opérations strictement plus prioritaires que l'opération `op` en

cours de construction (on parle d'analyse par `priorités ascendantes`). À l'initialisation, on utilise une priorité `root_prio` qui est le niveau le plus bas.

```
and rec parse_expr p = parse_a_expr () |> expand_op p

and parse_op p e1 = match !current_token with
  | (PLUS | STAR | LT | AND as op) when (prio op > p) ->
     next();
     let e2 = parse_expr (prio op) in
     let e = match op with
      | PLUS -> Bop(Add, e1, e2)
      | STAR -> Bop(Mul, e1, e2)
      | LT   -> Bop(Lt, e1, e2)
      | AND  -> Bop(And, e1, e2)
      | _    -> assert false
     in
     parse_op p e
  | _ -> e1

and parse_a_expr () = match take_token() with
  | LPAR -> let e = parse_expr root_prio in
            expect RPAR;
            e
  | NUM n -> Cst n
  | VAR x -> Var x
  | _ -> failwith "syntax error"
in
```

Enfin, on analyse un programme complet comme une séquence d'instructions, en vérifiant en plus que l'intégralité de l'entrée a bien été consommée.

```
let prog = parse_instrs () in
if !current_token = EOI then
  prog
else
  failwith "syntax error: unparsed tail"
```

3.5 Approfondissement : présentation alternative des dérivations

Alternativement, la dérivation d'une phrase selon une grammaire (T, N, S, R) donnée peut être caractérisée en utilisant, plutôt qu'un arbre, une suite de transformations de phrases partant du symbole de départ S et remplaçant progressivement des occurrences d'un symbole non terminal X par une séquence $a_1 \dots a_k$ telle que $X \prec a_1 \dots a_k$ est une règle dans R. Autrement dit, on *expanse* X. Dans ce cadre, on parle de ***grammaire générative*** et les règles sont appelées ***règles de production***.

Dérivation pour une grammaire générative. Formellement, étant donnée une grammaire (T, N, S, R) et m, m' deux séquence de symboles de $T \cup N$, on dit que u se ***dérive*** en v, et on note $u \to v$, si on a deux décompositions

$$\begin{aligned} m &= m_1 X m_2 \\ m' &= m_1 a_1 \dots a_n m_2 \end{aligned}$$

avec $X \in N$ et $X \prec a_1 \dots a_k \in R$.

Avec notre exemple de grammaire arithmétique simple, $E*(E)$ se dérive par exemple en $E*(E+E)$, en prenant

$$\begin{aligned} m_1 &= E*(\\ m_2 &=) \\ X &= E \\ \beta &= E+E \end{aligned}$$

De manière générale, on appelle ***dérivation*** une suite

$$m_1 \to m_2 \to \dots \to m_n$$

de telles transformations, qu'on note encore

$$m_1 \to^* m_n$$

Par exemple :

$$\begin{aligned} E &\to E * E \\ &\to \mathtt{n}_1 * E \\ &\to \mathtt{n}_1 * (E) \\ &\to \mathtt{n}_1 * (E + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + \mathtt{n}_3) \end{aligned}$$

On dit qu'une séquence m est ***dérivable*** lorsqu'il existe une dérivation $S \to^* m$. On parle de ***dérivation gauche*** lorsque, comme ici, on expanse à chaque étape le symbole non terminal le plus à gauche. Remarquez que, lorsque l'on arrive à une séquence ne contenant plus que des symboles terminaux, il n'y a plus de dérivation possible.

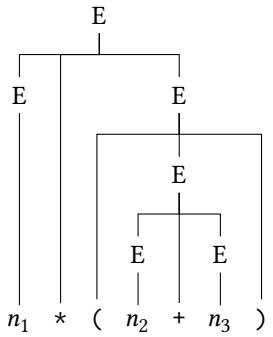
Équivalence entre les deux notions de dérivabilité. Les deux notions de dérivabilité sont équivalentes. Que l'on considère des arbres de dérivation ou des suites de transformations, les mêmes phrases sont dérivables à partir des mêmes symboles.

La nuance entre les deux formalismes se trouve dans les différentes manières de dériver une même phrase : une séquence de transformations est associée à un ordre dans lequel expanser les différents symboles non terminaux, alors qu'un arbre de dérivation ne retient qu'une vision structurée de la manière dont chaque symbole non terminal est expansé, sans spécifier d'ordre dans lequel les règles sont appliquées. Autrement dit, un arbre de dérivation décrit potentiellement plusieurs séquences de dérivation différentes, qui peuvent varier quant à l'ordre dans lequel sont appliquées les différentes règles. En revanche, toutes les séquences de dérivation associées à un arbre donné conservent la même structure, et les mêmes associations entre symboles non terminaux et groupes de symboles de la séquence produite.

Ci-dessous, les deux séquences de transformation correspondent au même arbre de dérivation.

$$\begin{aligned} E &\to E * E \\ &\to \mathtt{n}_1 * E \\ &\to \mathtt{n}_1 * (E) \\ &\to \mathtt{n}_1 * (E + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + \mathtt{n}_3) \end{aligned}$$

$$\begin{aligned} E &\to E * E \\ &\to E * (E) \\ &\to E * (E + E) \\ &\to E * (E + \mathtt{n}_3) \\ &\to E * (\mathtt{n}_2 + \mathtt{n}_3) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + \mathtt{n}_3) \end{aligned}$$



Montrons que pour tout arbre de dérivation d'une phrase m à partir d'un symbole a, on peut construire une séquence de dérivation de m à partir de a. On procède par récurrence sur la structure de l'arbre.

- Cas d'un arbre réduit à un unique nœud étiqueté par le symbole a, où la phrase m est a : on a la séquence de dérivation vide (ce cas couvre notamment la situation où a est un symbole terminal).
- Cas d'un arbre formé d'une racine X, dont les sous-arbres dérivent des mots m_1 à m_k à partir respectivement des symboles a_1 à a_k, où $X \prec a_1 \dots a_k$ est une règle de la grammaire. Par hypothèse d'induction il existe des séquences de dérivation de a_i à m_i pour tout $i \in [1; k]$. On peut alors par exemple construire la séquence de dérivation gauche suivante

  $$X \to a_1 a_2 \dots a_k \to^* m_1 a_2 \dots a_k \to^* m_1 m_2 \dots a_k \to^* \dots \to^* m_1 m_2 \dots m_k$$

  pour aller de X à $m_1 \dots m_k$.

Montrons à l'inverse que pour toute séquence de dérivation $a \to^* m$ d'une phrase m à partir d'un symbole a, il existe un arbre de dérivation de m à partir de a. On procède par récurrence sur la longueur de la séquence de dérivation.

- Cas d'une séquence de longueur zéro, où $a = m$, on conclut avec l'arbre comportant un unique nœud étiqueté par a.
- Cas d'une séquence $a \to m_1 \to \ldots \to m_k \to m_{k+1}$ de longueur $k+1$. Par hypothèse de récurrence il existe A un arbre de dérivation de m_k à partir de a. La dérivation $m_k \to m_{k+1}$ est faite avec des décompositions $m_k = u_1 X u_2$ et $m_{k+1} = u_1 a_1 \ldots a_i u_2$ pour une règle $X \prec a_1 \ldots a_i$. Autrement dit, en notant l le nombre de symbole dans u_1, la $l+1$-ème feuille de A porte le symbole X. On crée un arbre de dérivation pour m_{k+1} en donnant à cette feuille des fils, qui sont de nouvelles feuilles étiquetées par les symboles de la séquence $a_1 \ldots a_i$.

3.6 Approfondissement : construction d'un analyseur

La clé de la construction d'un analyseur récursif descendant est l'identification des règles d'expansion à choisir en fonction du prochain lexème de l'entrée. Pour cela on analyse la grammaire afin de trouver, pour chaque membre droit β de règle $X \prec \beta$, les symboles terminaux susceptibles de se trouver en tête d'une séquence dérivée de β. Pour obtenir un analyseur déterministe, on espère ne pas avoir plusieurs règles pour un même symbole X susceptibles de dériver des séquences commençant par un même symbole terminal.

Premiers et annulables. Considérons une grammaire (T, N, S, R), et une séquence $\alpha \in (T \cup N)^*$ de symboles terminaux ou non terminaux. On note $\mathsf{Premiers}(\alpha)$ l'ensemble des symboles terminaux qui peuvent se trouver en tête d'une séquence dérivée à partir de α.

$$\mathsf{Premiers}(\alpha) = \{\, a \in T \mid \exists \beta, \alpha \to^* a\beta \,\}$$

Si α a déjà la forme $a\beta$, avec $a \in T$ un symbole terminal, alors toute séquence dérivée aura encore cette forme, et a est l'unique symbole pouvant se trouver en tête. La situation est plus subtile lorsque α a la forme $X\beta$, avec $X \in N$ un symbole non terminal :

- d'une part, tout premier symbole de X est un premier possible de $X\beta$,
- d'autre part, *dans le cas où il est possible de dériver ε à partir de X*, les premiers de β sont également susceptibles de se trouver en tête.

Pour une analyse complète de la notion de premiers, nous caractérisons donc également l'ensemble des annulables, c'est-à-dire des symboles non terminaux à partir desquels il est possible de dériver la séquence vide.

$$\mathsf{Annulables} = \{\, X \in N \mid X \to^* \varepsilon \,\}$$

Caractérisation de l'ensemble Annulables. Le symbole X est annulable :

- s'il existe une règle $X \prec \varepsilon$, ou
- s'il existe une règle $X \prec X_1 \ldots X_k$ où tous les X_i sont des symboles non terminaux qui sont également annulables.

Caractérisation de l'ensemble $\mathsf{Premiers}(\alpha)$, pour α une séquence de $(T \cup N)^*$. On a les équations suivantes :

$$\begin{aligned}
\mathsf{Premiers}(\varepsilon) &= \emptyset \\
\mathsf{Premiers}(a\beta) &= \{a\} \\
\mathsf{Premiers}(X) &= \textstyle\bigcup_{X \prec \beta} \mathsf{Premiers}(\beta) \\
\mathsf{Premiers}(X\beta) &= \begin{cases} \mathsf{Premiers}(X) & \text{si } X \notin \mathsf{Annulables} \\ \mathsf{Premiers}(X) \cup \mathsf{Premiers}(\beta) & \text{si } X \in \mathsf{Annulables} \end{cases}
\end{aligned}$$

Dans un cas comme dans l'autre, on a des équations récursives. Pour trouver les ensembles des annulables et des premiers, on cherche donc des points fixes à ces équations.

Théorème de point fixe de Tarski. Si on considère un ensemble ordonné A fini et doté d'un plus petit élément $\bot$, et une fonction $F : A \to A$ croissante, alors :

- la fonction F admet au moins un point fixe, c'est-à-dire qu'il existe un élément $x \in A$ tel que $F(x) = x$, et
- en itérant F à partir de $\bot$, c'est-à-dire en calculant $F(\bot)$, puis $F(F(\bot))$, puis $F(F(F(\bot)))$, etc, on finit nécessairement par trouver un point fixe de F (et même plus précisément : le plus petit des points fixes).

Calcul des annulables. On cherche un ensemble de symboles non terminaux. On considère donc l'ensemble des parties de N, ordonné par l'ordre d'inclusion. Le plus petit élément est $\emptyset$. La fonction croissante qui nous intéresse est $F : \mathscr{P}(N) \to \mathscr{P}(N)$ définie par

$$F(E) = \{ X \mid X \to \varepsilon \} \cup \{ X \mid X \to X_1 \dots X_k,\ X_i \in E \}$$

Avec notre exemple des expressions arithmétiques nous avons : $F(\emptyset) = \{ E', M' \}$ et $F(\{ E', M' \}) = \{ E', M' \}$. Le point fixe est donc l'ensemble $\{ E', M' \}$, et les deux symboles E' et M' sont donc les deux seuls symboles non terminaux annulables. Ce calcul est souvent présenté dans un tableau de booléens donnant les annulables trouvés à chaque itération (on s'arrête lorsque deux lignes consécutives sont identiques).

	E	E'	M	M'	A
0.	F	F	F	F	F
1.	F	V	F	V	F
2.	F	V	F	V	F

Calcul des premiers. On cherche, pour chaque symbole non terminal, un ensemble de symboles terminaux. On considère donc l'ensemble $\mathscr{P}(T) \times \dots \times \mathscr{P}(T)$ (avec une composante par symbole non terminal), ordonné par le produit de l'ordre inclusion. Son plus petit élément est $(\emptyset, \dots, \emptyset)$, et la fonction croissante utilisée est celle qui recalcule les informations pour chaque symbole non terminal en fonction des informations déjà connues.

On présente généralement ce calcul dans un tableau, où chaque colonne correspond à un symbole non terminal, et chaque ligne à une itération. Le calcul s'arrête lorsque deux lignes deviennent identiques.

	E	E'	M	M'	A
0.	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\emptyset$	$\{+\}$	$\emptyset$	$\{*\}$	$\{n, (\}$
2.	$\emptyset$	$\{+\}$	$\{n, (\}$	$\{*\}$	$\{n, (\}$
3.	$\{n, (\}$	$\{+\}$	$\{n, (\}$	$\{*\}$	$\{n, (\}$
4.	$\{n, (\}$	$\{+\}$	$\{n, (\}$	$\{*\}$	$\{n, (\}$

Suivants. L'ensemble Premiers permet de caractériser une règle $X \prec \beta$ à appliquer lorsque le prochain symbole de l'entrée est le premier symbole de la séquence dérivée à partir de β. Mais que faire alors des règles de la forme $X \prec \varepsilon$? Dans une telle règle il n'existe pas de symbole pouvant se trouver en tête. On l'applique lorsque le prochain lexème de l'entrée n'est pas le premier symbole de X, mais plutôt le premier symbole *suivant* X.

À nouveau en analysant la grammaire, on peut caractériser l'ensemble des ***suivants*** d'un symbole non terminal $X \in N$, c'est-à-dire les symboles terminaux qui peuvent apparaître immédiatement après X dans une phrase dérivée à partir du symbole de départ.

$$\text{Suivants}(X) = \{ a \in T \mid \exists \alpha, \beta,\ S \to^* \alpha X a \beta \}$$

Dans la grammaire des expressions arithmétiques, le symbole + appartient aux suivants de M, puisque nous avons par exemple la dérivation

$$S \to M\, E' \to M + E$$

mais également aux suivants de A puisque nous avons également la dérivation

$$S \to M\, E' \to M + E \to A\, M' + E \to A\, \varepsilon + E = A + E$$

Pour calculer l'ensemble des suivants de X, on regarde toutes les apparitions de X dans un membre droit de règle. Pour chaque règle de la forme $Y \prec \alpha X \beta$, on inclut dans $\text{Suivants}(X)$:

- les symboles terminaux susceptibles d'apparaître en tête d'une séquence dérivée à partir de β, c'est-à-dire $\text{Premiers}(\beta)$,
- dans le cas où il est possible de dériver la séquence vide à partir de β, tous les suivants de Y peuvent également suivre X (note : cela couvre le cas où β *est* la séquence vide, mais aussi le cas où β est une séquence de symboles non terminaux annulables).

On a à nouveau des équations récursives, à résoudre par la même technique que précédemment.

Calcul des suivants. On cherche, pour chaque symbole non terminal, un ensemble de symboles terminaux. La mise en place est donc la même que pour le calcul des premiers. À la première itération, on voit apparaître les symboles suivants qui sont directement visibles dans une règle. Dans les itérations suivantes, les informations se propagent. On considère en outre que le symbole terminal spécial # de fin d'entrée (correspondant à `EOF`) appartient aux suivants du symbole de départ S.

Dans notre exemple des expressions arithmétiques :

$$\begin{aligned}
\text{Suivants}(E) &= \{\#,)\} \cup \text{Suivants}(E') \\
\text{Suivants}(E') &= \text{Suivants}(E) \\
\text{Suivants}(M) &= \{+\} \cup \text{Suivants}(E) \cup \text{Suivants}(M') \\
\text{Suivants}(M') &= \text{Suivants}(M) \\
\text{Suivants}(A) &= \{*\} \cup \text{Suivants}(M)
\end{aligned}$$

Notamment, des informations vont se propager de E à M et de M à A, car M peut apparaître à la fin d'une séquence dérivée de E (après annulation de E'), et car A peut apparaître à la fin d'une séquence dérivée de M (après annulation de M').

	E	E'	M	M'	A
0.	$\{\#\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\{\#,)\}$	$\{\#\}$	$\{\#, +\}$	$\emptyset$	$\{*\}$
2.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +\}$	$\{\#, +, *\}$
3.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +,)\}$	$\{\#, +, *,)\}$
4.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +,)\}$	$\{\#, +, *,)\}$

Analyse LL. Une fois calculés les premiers et les suivants, on peut construire une ***table d'analyse LL(1)*** indiquant, pour chaque paire d'un symbole non terminal X en cours d'analyse et d'un symbole terminal a, la règle à appliquer lorsque l'on tente de reconnaître un fragment X et que le prochain lexème de l'entrée est a.

Le principe de construction est le suivant :

— Pour chaque règle $X \prec \beta$ avec $\beta \neq \varepsilon$ et chaque symbole terminal $a \in \text{Premiers}(\beta)$ on indique la règle $X \prec \beta$ dans la case de la ligne X et de la colonne a.

— Pour chaque règle $X \prec \varepsilon$ et chaque symbole terminal $a \in \text{Suivants}(X)$ on indique la règle $X \prec \varepsilon$ dans la case de la ligne X et de la colonne a.

Si chaque case contient au plus une règle, on obtient une table d'analyse déterministe. La grammaire utilisée, qui est donc compatible avec cette technique d'analyse, est dite ***LL(1)*** (***L**eft-to-right scanning, **L**eftmost derivation, using **1** look-ahead symbol*).

C'est le cas par exemple pour la grammaire des expressions arithmétiques dont nous avons calculé les premiers et suivants, pour laquelle nous avons la table d'analyse LL(1) suivante.

	n	+	*	(	)	#
E	$M\,E'$			$M\,E'$		
E'		$+\;E$			ε	ε
M	$A\,M'$			$A\,M'$		
M'		ε	$*\;M$		ε	ε
A	n			$(\;E\;)$		

Si à l'inverse, une case de la table contient plusieurs règles, on dit que la table contient un ***conflit***, et que la grammaire *n'est pas* LL(1). Pour obtenir un analyseur dans ce cas, il faut revoir la grammaire ou utiliser une autre technique.

Notez qu'être ou non compatible avec l'analyse LL(1) est une propriété de la grammaire, et non du langage décrit par la grammaire. Par exemple, la dernière grammaire que nous avons utilisée pour les expressions arithmétiques est LL(1), mais aucune des précédentes ne l'était.

Bilan sur l'analyse récursive descendante. La technique vue à cette section est très facile à programmer, à condition que la grammaire ait une forme compatible. C'est le cas notamment lorsque chaque construction du langage est introduite par un mot-clé ou un symbole spécifique (par exemple : **`if`** ou **`while`**). Dans les autres situations cependant, transformer la grammaire de notre langage de manière à la rendre compatible avec l'analyse LL(1) peut être compliqué, et le résultat obtenu peut être cryptique, et éloigné de la manière naturelle de décrire le langage.

D'autres techniques existent, avec des caractéristiques différentes.

4 Traduction en assembleur d'un langage impératif

La compilation est un processus en plusieurs étapes. Nous nous sommes concentrés au chapitre précédent sur une étape d'***analyse***, visant à extraire la structure et le sens du programme source reçu en entrée. Ensuite vient une phase de ***synthèse***, qui part de l'arbre de syntaxe abstraite d'un programme source, et vise à produire un programme cible équivalent au programme source. Le programme cible aura la forme d'un fichier écrit dans un certain langage cible, comme l'était le programme source avant analyse.

Le langage cible peut être par exemple :

- un autre langage de haut niveau, par exemple C ou javascript, et on pourra alors utiliser l'infrastructure de ce langage pour exécuter le programme obtenu,
- du ***bytecode***, c'est-à-dire du code bas niveau pour une machine virtuelle, par exemple les machines virtuelles de java, caml ou python,
- un langage ***assembleur***, pour exécution directe sur une architecture matérielle donnée, par exemple Intel x86, ARM, MIPS ou RISC-V (après assemblage et édition de liens).

Dans ce chapitre, nous partons d'un programme IMP déjà analysé et donné sous la forme d'un arbre de syntaxe abstraite, et le traduisons en un format adapté à l'exécution par un ordinateur réel.

4.1 Architecture cible

La cible ultime de la compilation est la production d'un programme directement exécutable par un ordinateur physique. La nature d'un tel programme est intimement liée à l'architecture de l'ordinateur cible.

Architecture et boucle d'exécution. Dans ce cours on utilisera une architecture simple mais réelle appelée MIPS, qui est un précurseur de l'architecture moderne RISC-V. Les deux principaux composants de l'architecture sont :

- un *processeur*, comportant un petit nombre de registres qui permettent de stocker quelques données directement accessibles, ainsi que des unités de calcul qui opèrent sur les registres, et
- une grande quantité de *mémoire*, où sont stockées à la fois des données et le programme à exécuter lui-même.

On a deux unités de base pour la mémoire : l'*octet*, une unité universelle désignant un groupe de 8 bits, et le ***mot mémoire***, qui représente l'espace alloué à une donnée « ordinaire », comme un nombre entier. Dans notre architecture, le mot mémoire vaut 4 octets : les données manipulées sont donc généralement représentées par des séquences de 32 bits. En conséquence, un ***entier machine***, c'est-à-dire un nombre entier primitif manipulé par ce processeur est compris entre -2^{31} et $2^{31}-1$ (ou entre 0 et $2^{32}-1$ dans le cas d'entiers « non signés »).

La plupart des architectures majeures existent en versions 32 bits et 64 bits.

L'architecture MIPS comporte 32 registres, pouvant chacun stocker un mot mémoire. Ces données stockées dans les registres sont directement accessibles à l'unité de calcul.

La mémoire principale a une étendue de 2^{32} octets. Chaque octet de la mémoire est associé à une ***adresse***, qui est un entier entre 0 et $2^{32}-1$, et c'est exclusivement en utilisant ces adresses que l'on accède aux éléments stockés en mémoire. Les données étant organisées sur des mots mémoire de 4 octet, l'architecture impose en outre que les adresses des données ordinaires soient des multiples de 4 (il existe quelques exceptions). L'accès à la mémoire est relativement coûteux : une seule opération de lecture ou d'écriture en mémoire a un coût nettement supérieur à une opération arithmétique travaillant directement sur les registres.

Le programme à exécuter est une séquence d'instructions directement interprétables par le processeur. Chaque instruction est codée sur 4 octets, et l'ensemble est stocké de manière contiguë dans la mémoire.

Un registre spécial `pc` (qui ne fait pas partie des 32) contient l'adresse de l'instruction courante (*program counter*, ou *pointeur de code*). L'exécution d'un programme procède en répétant le cycle suivant :

1. lecture d'un mot mémoire à l'adresse `pc` (*fetch*),
2. interprétation des octets lus comme une instruction (*decode*),
3. exécution de l'instruction reconnue (*execute*),
4. mise à jour de `pc` pour passer à l'instruction suivante (par défaut : incrémenter de 4 octets pour passer au mot mémoire suivant).

Instructions. On distingue trois catégories principales d'instructions :

— des instructions arihtmétiques, appliquant des opérations élémentaires à des valeurs stockées dans des registres,
— des instructions d'accès à la mémoire, pour transférer des valeurs de la mémoire vers les registres ou inversement,
— des instructions de contrôle, pour gérer le pointeur de code pc.

Dans les 32 bits utilisés pour coder une instruction, les 6 premiers désignent l'opération à effectuer (***opcode***), et les suivants donnent les paramètres ou des précisions éventuelles sur l'opération. Ci-dessous, quelques exemples avant d'aborder à la section suivante la manière de programmer avec ces instructions.

Par exemple : l'instruction numéro 9 prend en paramètres un registre source r_s, un registre de destination r_t, et un nombre n non signé de 16 bits, et place dans r_t la valeur $r_s + n$. Il suffit de 5 bits pour désigner l'un des 32 registres. Si r_s est le registre numéro 5, r_t le registre numéro 17, et n la valeur 42, alors l'instruction est représentée par un mot mémoire décomposé ainsi :

op	r_s	r_t	n
001000	00101	10001	0000000000101010

Autre exemple : l'instruction numéro 15 prend en paramètres un registre de destination r_t et un nombre n de 16 bits, et place n dans les deux octets supérieurs de r_t. Si r_t est le registre numéro 17 et n la valeur 42, alors l'instruction est représentée par le mot mémoire suivant, où 5 bits ne sont pas utilisés.

op		r_t	n
001111	00000	10001	0000000000101010

Quizz : comment utiliser les instructions précédentes pour charger la valeur `0xeff1cace` *dans le registre numéro 2 ?*

La plupart des opérations arithmétiques binaires utilisent l'*opcode* 0, l'opération précise étant alors donnée par les 6 derniers bits de l'instruction (cette dernière partie est appelée la *fonction*). Par exemple, l'opération d'addition est identifiée par l'*opcode* 0 et la fonction 32. Ces instructions prennent invariablement trois paramètres : deux registres r_s et r_t pour les valeurs auxquelles appliquer l'opération, et un registre de destination r_d. Ci-dessous, opération d'addition avec pour r_s, r_t et r_d les registres de numéros respectifs 1, 2 et 3. Notez que 5 bits sont inutilisés.

op	r_s	r_t	r_d		f
000000	00001	00010	00011	00000	100000

Certaines opérations ignorent le paramètre r_s et fournissent à la place une donnée constante à l'aide des 5 bits qui étaient restés libres ci-dessus. Ainsi, l'opération de décalage vers la gauche prend trois paramètres : un registre r_t à qui appliquer un décalage, un registre r_d de destination, et une constante k sur 5 bits indiquant de combien de bits vers la gauche décaler r_t. C'est opération correspond toujours à l'*opcode* 0, mais cette fois avec la fonction 0. Ainsi, pour décaler de le registre numéro 4 de 5 bits vers la gauche et placer le résultat dans le registre numéro 6, on a :

op		r_t	r_d	k	f
000000	00000	00100	00110	00101	000000

Ce schéma général de découpage vaut encore pour les instructions d'accès à la mémoire. L'instruction de lecture d'un mot mémoire par exemple prend en paramètres deux registres r_s et r_t et un entier signé n sur 16 bits. Le registre r_s est supposé contenir une adresse, et l'entier n est appelé *décalage*. L'instruction de lecture calcule une adresse a en additionnant l'adresse de base r_s et le décalage n, et transfère vers le registre r_t le mot mémoire lu à l'adresse a. Le code de cette opération est 35. Ainsi, pour placer dans le registre numéro 3 la donnée trouvée à l'adresse obtenue en ajoutant 8 octets à l'adresse donnée par le registre numéro 5, on a la séquence :

op	r_s	r_t	n
100011	00101	00011	0000000000001000

Les instructions de contrôle utilisent encore une variante de ce même découpage. On a par exemple une instruction de saut conditionnel, qui prend en paramètres un registre r_s et

un entier n signé sur 16 bits, et qui avance de n instructions dans le code du programme si la valeur de r_s est strictement positive. Le code de cette instruction est 7. Pour reculer de 5 instructions lorsque la valeur du registre 3 est strictement positive, on a donc l'instruction :

op	r_s	r_t	n
000111	00011	00000	1111111111111011

Enfin, l'instruction numéro 2 prend pour unique paramètre un entier sur 26 bits interprété comme l'adresse a d'une instruction i, et va faire se poursuivre l'exécution à partir de cette instruction i. Autrement dit, on écrase la valeur du pointeur `pc` pour la remplacer par a. Pour aller à l'instruction d'adresse `0x00efface` on aura donc l'instruction :

op	a
000010	00111011111111101011001110

Nous verrons à la section suivante qu'en pratique, on ne compose pas directement ces codes représentant chaque instruction. On utilise à la place une forme textuelle appelée le ***langage assembleur***, qui est ensuite automatiquement traduite en la version codée par des mots binaires.

Autres familles d'architectures. De nombreux aspects de l'architecture MIPS que nous venons de décrire sont partagés avec les autres architectures majeures. Pour commencer, la boucle d'exécution *fetch/decode/execute* avec un programme stocké en mémoire est une réalisation directe du modèle de von Neumann, à la base de tous les ordinateurs depuis l'origine de l'informatique.

D'une famille d'architectures à l'autre, on observe des variations par exemple sur la taille d'un mot mémoire, avec notamment les architectures 64 bits où le mot mémoire est de 8 octets. Le nombre de registres peut également varier, ainsi que l'ensemble des instructions disponibles ou les manières d'accéder à la mémoire. Les différences les plus significatives ne sont en revanche pas toujours les plus visibles pour le programmeur.

Dans l'architecture MIPS, les instructions sont codées suivant un schéma uniforme qui limite le nombre d'instructions qui peuvent être proposées. Cette approche est typique des architectures de la famille *RISC* (*Reduced Instruction Set Computer*), qui comprennent notamment les architectures ARM et RISC-V.

À l'inverse, les architectures de la famille *CISC* (*Complex Instruction Set Computer*) proposent un codage variable des instructions, par exemple allant de 1 à 8 octets. Ceci permet entre autres choses de proposer un nombre beaucoup plus importants d'instructions. Cette famille comprend notamment les architectures Intel.

Dans une représentation variable de type *CISC*, on dispose d'instructions riches permettant un code optimisé. En outre, le format variable permet de compresser le code, en donnant une représentation compacte aux instructions les plus fréquemment utilisées. Dans une représentation uniforme de type *RISC*, le décodage des instructions est plus simple, et l'ensemble du processeur est de même plus simple et utilise moins de transistors, ce qui permet une moindre consommation d'énergie. En conséquence les architectures ARM sont omniprésentes dans les *smartphones*, où la gestion de la batterie est critique. Les architectures Intel, réputées plus puissantes, ont à l'inverse longtemps été hégémoniques dans le domaine des ordinateurs grand public, mais cela pourrait basculer. En particulier, le format uniforme des architectures RISC facilite grandement le traitement de plusieurs instructions en parallèle, ce qui est une source très importante d'optimisation des architectures. La montée en puissance de ces optimisations rend maintenant les architectures RISC compétitives avec les architectures CISC en termes de puissance de calcul, et les puces ARM commencent à prendre une place dans le monde des ordinateurs grand public [4].

4.2 Langage assembleur MIPS

En pratique, on n'écrit pas directement les suites de bits du langage machine. On utilise à la place un langage d'assemblage, ou assembleur, qui est quasiment isomorphe au langage machine mais permet une écriture plus agréable. En langage assembleur on a en particulier :

— des écritures textuelles pour les instructions,

4. Ce commentaire est écrit en 2021. Apple est en train de remplacer les processeurs Intel par des processeurs ARM dans l'ensemble de ses ordinateurs, avec succès jusque là (gros gain énergétique sans perte notable de puissance de calcul), et des rumeurs suggèrent que Google préparerait une évolution similaire.

— la possibilité d'utiliser des étiquettes symboliques plutôt que des adresses explicites,
— une allocation statique des données globales,
— quelques *pseudo*-instructions, qui correspondent à des combinaisons simples d'instructions réelles.

Dans cette section, on présente le langage assembleur MIPS.

Pour tester nos programmes MIPS, on utilisera le simulateur MARS. Ce simulateur peut être appelé directement depuis la ligne de commande pour exécuter un programme assembleur donné, mais dispose aussi d'un mode graphique dans lequel on peut suivre pas à pas l'exécution d'un programme et l'évolution des registres et de la mémoire.

L'architecture RISC-V, bien que modernisée par rapport à MIPS, utilise en fait un langage assembleur quasiment identique. Du point de vue du programmeur, ce cours vaut donc *presque* pour les deux architectures.

Registres. Dans l'assembleur MIPS, les 32 registres sont désignés par leur numéro, de `$0` jusqu'à `$31`. Alternativement à son numéro, chaque registre possède en outre un nom reflétant sa fonction.

n°	*nom*
`$0`	`$zero`
`$1`	`$at`
`$2`	`$v0`
`$3`	`$v1`
`$4`	`$a0`
`$5`	`$a1`
`$6`	`$a2`
`$7`	`$a3`

n°	*nom*
`$8`	`$t0`
`$9`	`$t1`
`$10`	`$t2`
`$11`	`$t3`
`$12`	`$t4`
`$13`	`$t5`
`$14`	`$t6`
`$15`	`$t7`

n°	*nom*
`$16`	`$s0`
`$17`	`$s1`
`$18`	`$s2`
`$19`	`$s3`
`$20`	`$s4`
`$21`	`$s5`
`$22`	`$s6`
`$23`	`$s7`

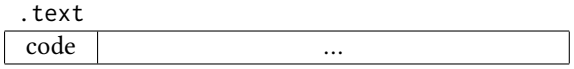
n°	*nom*
`$24`	`$t8`
`$25`	`$t9`
`$26`	`$k0`
`$27`	`$k1`
`$28`	`$gp`
`$29`	`$sp`
`$30`	`$fp`
`$31`	`$ra`

Les 24 registres `$v*`, `$a*`, `$t*` et `$s*` sont des registres ordinaires, que l'on peut librement utiliser pour des calculs (on verra les spécificités de ces quatre familles plus tard). Les 8 autres registres ont des rôles particuliers : `$gp`, `$sp`, `$fp` et `$ra` contiennent des adresses utiles, `$zero` contient toujours la valeur 0 et ne peut pas être modifié, et les 3 registres `$at` et `$k*` sont réservés respectivement pour l'assembleur et pour le système (il ne faut donc pas les utiliser).

Instructions et pseudo-instructions. Un programme en langage assembleur MIPS prend la forme d'un fichier texte, où chaque ligne contient une instruction ou une annotation. On introduit une liste d'instructions avec l'annotation

```
.text
```

Les instructions ainsi introduites sont placées dans une zone de la mémoire réservée au programme, tout en bas de la mémoire (c'est-à-dire aux adresses les plus basses).



Chaque instruction est construite avec un mot clé appelé ***mnémonique***, désignant l'opération à effectuer, et un certain nombre de paramètres séparés par des virgules. Voici par exemple une instruction qui initialise le registre `$t0` avec la valeur 42

```
li $t0, 42
```

et une instruction qui copie la valeur présente dans le registre `$t0` vers le registre `$t1`.

```
move $t1, $t0
```

Comme le montrent ces exemples, les paramètres donnés aux instructions peuvent être, selon les instructions, des registres désignés par leur nom et/ou des constantes entières écrites de manière traditionnelle.

Une mnémonique décrit l'opération à effectuer, en général à l'aide d'un mot ou d'un acronyme. On a ici `move` pour le « déplacement » d'une valeur ou `li` pour *Load Immediate*, c'est-à-dire pour le chargement d'une valeur constante (on appelle valeur *immédiate* une valeur constante directement fournie dans le code assembleur). Certaines de ces mnémoniques correspondent directement à des instructions machine, et peuvent être associées à un *opcode* et le cas échéant à une fonction. D'autres mnémoniques sont des *pseudo-instructions*, c'est-à-dire des raccourcis pour de courtes séquences d'instructions machine (généralement des séquences d'une ou deux instructions seulement).

En l'occurrence :

— `li` est une pseudo-instruction qui s'adapte à la constante à charger : si cette constante s'exprime sur 16 bits alors elle sera traduite par une unique instruction machine, mais dans le cas contraire on aura deux instructions pour charger indépendamment les deux octets hauts et les deux octets bas. Ainsi, cette pseudo-instruction permet au programmeur de s'affranchir de la limite de 16 bits pour les valeurs immédiates.
— `move` est une pseudo-instruction, traduite par une unique instruction d'addition : plutôt que d'inclure dans les circuits du processeur une instruction spécifique pour une affectation de la forme `$k` ← `$k'`, on réutilise l'instruction d'addition qui est de toute façon présente, en fixant le deuxième opérande à zéro : `$k` ← (`$k'` + `$zero`).

Arithmétique. Les opérations arithmétiques et logiques du langage assembleur MIPS suivent toutes le même format « trois adresses » : après la mnémonique identifiant l'opération viennent dans l'ordre le registre cible (où sera placé le résultat de l'opération) puis les deux opérandes.

```
<mnemo>  <dest>, <r1>, <r2>
```

On y trouve des opérations variées, dont la plupart vous seront familières.
— Opérations arithmétiques : `add`, `sub`, `mul`, `div`, `rem` (*remainder* : reste), ...
— Opérations logiques : `and`, `or`, `xor` (*exclusive or*), ...
— Comparaisons : `seq` (*equal* : égalité), `sne` (*not equal* : inégalité) `slt` (*less than* : $<$), `sle` (*less or equal* : $\leqslant$), `sgt` (*greater than* : $>$), `sge` (*greater or equal* : $\geqslant$), ...

À ces opérations binaires s'ajoutent quelques opérations unaires classiques également, avec un registre de destination et un seul opérande : `abs` (valeur absolue), `neg` (opposé), **`not`** (négation logique).

Les opérations arithmétiques admettent parfois des variantes pour l'arithmétique non signée, c'est-à-dire pour manipuler des entiers entre 0 et $2^{32} - 1$ plutôt que des entiers entre -2^{31} et $2^{31} - 1$. Ces opérations sont repérables par la présence d'un `u`. Par exemple : `addu`.

Un certain nombre des opérations précédentes admettent également une variante dans laquelle le deuxième opérande est une valeur immédiate plutôt qu'un registre. Ces opérations sont repérables par la présence d'un `i`. Par exemple : `addi`, ou encore `addiu`.

Enfin, les opérations de manipulation des séquences de bits par décalage ou rotation suivent le même format, mais en prenant par défaut une valeur immédiate pour le deuxième opérande (l'amplitude du décalage).
— Décalages et rotations : `sll` (*shift left logical*), `sra` (*shift right arithmetic*), `srl` (*shift right logical*), `rol` (*rotation left*), `ror` (*rotation right*), ...

Des variantes avec une amplitude de décalage variable existent, repérées par la présence d'un `v`. Par exemple : `sllv`. Dans ce cas, le deuxième opérande est un registre.

Voici par exemple une séquence d'instructions pour évaluer la comparaison `3*4 + 5 < 2*9`, ainsi qu'un tableau montrant l'évolution des différents registres à mesure que l'exécution progresse.

		`$t0`	`$t1`	`$t2`	`$t3`	`$t4`	`$t5`	`$t6`	`$t7`	`$t8`
`li`	`$t0, 3`	3								
`li`	`$t1, 4`	3	4							
`li`	`$t2, 5`	3	4	5						
`li`	`$t3, 2`	3	4	5	2					
`li`	`$t4, 9`	3	4	5	2	9				
`mul`	`$t5, $t0, $t1`	3	4	5	2	9	12			
`add`	`$t6, $t5, $t2`	3	4	5	2	9	12	17		
`mul`	`$t7, $t3, $t4`	3	4	5	2	9	12	17	18	
`slt`	`$t8, $t6, $t7`	3	4	5	2	9	12	17	18	1

L'exemple précédent n'est cependant guère judicieux : il utilise un nouveau registre pour chaque opération, et n'utilise pas toujours les opérations les plus efficaces. Voici une meilleure

version pour réaliser le même calcul.

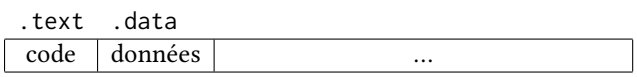
	$t0	$t1
`li $t0, 3`	3	
`li $t1, 4`	3	4
`mul $t0, $t0, $t1`	12	4
`addi $t0, $t0, 5`	17	4
`li $t1, 9`	17	9
`sll $t1, $t1, 1`	17	18
`slt $t0, $t0, $t1`	1	18

Données statiques. À côté des instructions, un programme assembleur peut également déclarer et initialiser un certain nombre de données, correspondant par exemple à des variables globales du programme.

On introduit une liste de déclarations de telles données avec l'annotation

```
.data
```

En mémoire, les données ainsi déclarées sont placées dans une zone située après la zone réservée aux instructions du programme.



La définition d'une donnée ou d'un groupe de données combine :

— la déclaration d'une étiquette symbolique, c'est-à-dire d'un nom qui pourra être utilisé pour accéder à la donnée,

— la fourniture d'une ou plusieurs valeurs initiales.

Un mot-clé permet de préciser la nature des données fournies : `.word` pour une valeur 32 bits (4 octets), `.byte` pour une valeur d'un unique octet (8 bits), `.asciiz` pour une chaîne de caractères au format ASCII (un caractère par octet, la chaîne étant terminée par l'octet zéro (`0x00`).

On déclare ainsi une donnée désignée par le nom `reponse` et valant 42 :

```
reponse: .word 42
```

Ainsi, un mot mémoire va être réservé dans la zone des données et initialisé avec la valeur 42. L'identifiant `reponse` désigne l'adresse de ce mot mémoire, et peut être utilisée notamment pour récupérer ou modifier la valeur qui y est stockée.

On peut de même introduire une séquence de données.

```
prems: .word 2 3 5 7 11 13 17 19
```

La première donnée est placée à l'adresse correspondant à l'étiquette `prems`, puis les suivantes sont l'une après l'autre, de 4 octets en 4 octets.

L'adresse réservée pour chaque donnée sera calculée au moment de la traduction du programme assembleur en langage machine. À cette occasion, chaque mention d'une étiquette sera remplacée en dur par un accès à l'adresse correspondante.

Accès à la mémoire. Le format standard des adresses mémoire en MIPS contient deux composantes :

— une adresse de base, donnée par la valeur d'un registre `$r`,

— un décalage, donné par une constante `d` (16 bits, signée).

On note `d($r)` une telle adresse, dont la valeur est donc `$r + d`.

Les instructions d'accès à la mémoire permettent alors de transférer une valeur depuis une adresse mémoire vers un registre (lecture, *load*) ou au contraire depuis un registre vers une adresse mémoire (écriture, *store*).

```
lw $t0, 8($a1)
sw $t0, 0($t1)
```

Les deux instructions précédentes enchaînent donc la lecture du mot mémoire à l'adresse obtenue en ajoutant 8 à la valeur du registre `$a1` (la valeur lue étant placée dans le registre `$t0`), puis l'écriture de la valeur du registre `$t0` à l'adresse donnée directement par la valeur du registre `$t1`.

Les mnémoniques `lw` et `sw` signifient respectivement *Load Word* et *Store Word*. Des variantes existent pour lire ou écrire seulement un octet (`lb`, `lbu`, `sb`), ou un demi-mot (`lh`, `lhu`, `sh`), ou encore un double mot (`ld`, qui transfère les 8 octets lus vers 2 registres : celui donné dans l'instruction et le suivant, et `sd` qui fonctionne symétriquement).

L'assembleur permet également quelques notations simplifiées des adresses, qui sont converties vers le format standard au moment de la traduction vers le langage machine. On peut par exemple omettre le décalage lorsqu'il vaut zéro

```
sw $t0, ($t1)
```

ou encore accéder directement à l'adresse donnée par une étiquette, avec un éventuel décalage positif ou négatif.

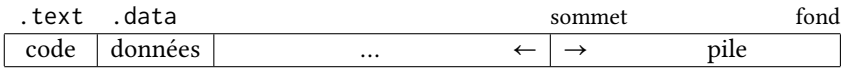
```
lw $t0, prems
lw $t1, prems+4
lw $t2, prems+12
```

Enfin, une instruction `la` (*Load Address*) permet de récupérer ou calculer une adresse, sans lire son contenu.

```
la $t0, prems
```

Pile. Le mécanisme des données statiques n'est pas suffisant dans un programme ordinaire [5]. On a généralement besoin de pouvoir stocker des données en mémoire, ne serait-ce que temporairement, sans avoir prédit précisément dès l'écriture du programme la quantité d'espace occupée. Ainsi, dans un programme ordinaire la plus grande partie de la mémoire est *dynamique*, c'est-à-dire que l'allocation de l'espace aux différentes données n'y est décidée que *pendant l'exécution* du programme, et qu'une adresse occupée par une donnée à un moment de l'exécution a vocation à être libérée lorsque la donnée n'est plus utile, pour être ensuite réaffectée à une nouvelle donnée.

La partie supérieure de cette grande région de mémoire dynamique est appelée la ***pile***, et fonctionne de manière identique à la structure de données de pile. Il s'agit donc d'une structure linéaire, dont une extrémité est appelée le ***fond*** et l'autre le ***sommet***, et qui est modifiée uniquement du côté de son sommet : on peut ajouter de nouveaux éléments au sommet de la pile, ou retirer les éléments du sommet. Cette organisation implique qu'un élément retiré de la pile est toujours l'élément de la pile le plus récemment ajouté.



Petite particularité notable ici : la pile est placée à l'extrémité de la mémoire, du côté des adresses les plus hautes. Il n'est donc pas possible de l'étendre « vers le haut ». Le fond de cette pile est donc calé sur l'adresse la plus grande de la mémoire, et le sommet à une adresse inférieure : la pile croît ainsi en s'étendant vers les adresses inférieures.

Le sommet de la pile est donc l'adresse mémoire la plus basse utilisée par la pile. Cette adresse est stockée dans le registre `$sp`. On réalise les opérations usuelles d'une structure de pile de la manière suivante.

— Pour consulter la valeur au sommet de la pile (peek), on lit un mot à l'adresse donnée par `$sp`.

```
    lw   $t0, 0($sp)
```

Pour aller consulter des éléments plus profonds dans la pile, on ajoute un décalage de 4 octets pour chaque élément à sauter. On consulte donc le quatrième élément avec un décalage de 12 octets.

```
    lw   $t0, 12($sp)
```

— Pour ajouter un élément au sommet de la pile (push), on décrémente `$sp` de 4 octets, puis on écrit la valeur souhaitée à la nouvelle adresse `$sp`.

```
    addi $sp, $sp, -4
    sw   $t0, 0($sp)
```

5. On s'astreint parfois à s'en contenter, pour limiter les possibilités de bugs dans des catégories particulières de programmes, notamment dans des domaines critiques comme l'avionique.

— Pour retirer l'élément au sommet de la pile, on incrémente `$sp` de 4 octets. Notez qu'il n'est pas nécessaire « d'effacer » la valeur présente à l'ancienne adresse `$sp` : le simple fait que cette adresse soit maintenant inférieure à `$sp` la désigne comme insignifiante.

```
    addi $sp, $sp, 4
```

L'opération usuelle pop, consistant à récupérer l'élément au sommet tout en le retirant de la pile, combine cet incrément avec une opération peek.

```
    lw   $t0, 0($sp)
    addi $sp, $sp, 4
```

Le code assembleur obtenu est d'une certaine manière symétrique à celui de push.

Sauts et branchements. Rappelons que les instructions d'un programme MIPS sont stockées en mémoire. Chaque instruction a donc une adresse. De même que les adresses des données, les adresses de certaines instructions peuvent être désignées par des étiquettes : une étiquette insérée dans une séquence d'instructions MIPS désigne l'adresse de l'instruction qui la suit immédiatement, et pourra être utilisée comme cible d'une instruction de saut.

MIPS propose une variété d'instructions (ou pseudo-instructions) de branchement conditionnel. En voici un exemple :

```
  beq  $t0, $t1, lab
```

Cette instruction compare les valeurs des registres `$t0` et `$t1`. Si ces deux valeurs sont égales, alors l'exécution se poursuivra avec l'instruction d'étiquette `lab` (techniquement, l'instruction de branchement commande une modification du registre spécial pc). Sinon, l'exécution se poursuivra naturellement avec l'instruction suivante.

Des variantes existent pour d'autres modalités de comparaison

$=$	$\neq$	$<$	$\leqslant$	$>$	$\geqslant$
beq	bne	blt	ble	bgt	bge

et chacune admet à nouveau une variante pour comparer une unique valeur à zéro : beqz, bnez, bltz, ... Certaines admettent également pour deuxième paramètre une valeur immédiate plutôt qu'un registre. Enfin, une pseudo-instruction `b lab` provoque un branchement inconditionnel vers l'instruction d'étiquette `lab`.

Voici un exemple de fragment de code calculant la factorielle du nombre stocké dans le registre `$a0`, et plaçant le résultat dans le registre `$v0`.

```
1          move $t0, $a0
2          li   $t1, 1
3          b    test
      loop:
4          mul  $t1, $t1, $t0
5          addi $t0, $t0, -1
      test:
6          bgtz $t0, loop
7          move $v0, $t1
```

Les deux premières instructions initialisent les deux registres `$t0` et `$t1` qui seront utilisés pour le calcul. Les instructions 4 à 6 forment une boucle, et la dernière instruction transfère le résultat dans `$v0` comme demandé. Le corps de la boucle est constitué des instructions 4 et 5, la première étant associée à l'étiquette `loop`. Le test de la boucle est à l'instruction 6, associée à l'étiquette `test`. Lorsque ce test est positif, on déclenche un nouveau tour de boucle à l'aide d'un branchement vers `loop`. À l'inverse, lorsque le test est négatif on poursuit avec l'instruction suivante, c'est-à-dire l'instruction 7. Enfin, l'instruction 3 démarre la boucle en branchant vers le test. *Notez que l'on aurait pu se passer de* `$t0`*, de* `$t1` *et des instructions 1 et 7 en travaillant exclusivement avec* `$a0` *et* `$v0`*.*

Les instructions de branchement sont utilisées pour des déplacements « locaux » dans la séquence d'instructions, c'est-à-dire en restant dans la même unité de code (par exemple : la même fonction). Les instructions machine correspondantes font avancer ou reculer le pointeur de code d'un certain nombre d'instructions, ne dépassant pas 2^{15}.

À l'inverse, les instructions de saut définissent la prochaine instruction de manière absolue, en fournissant directement son adresse. Cela peut se faire à l'aide d'une étiquette, et dans ce cas la différence avec une instruction de branchement n'est pas flagrante,

```
j    label
```

ou en fournissant directement une adresse calculée, stockée dans un registre.

```
jr   $t0
```

Les instructions de saut servent notamment dans le cadre d'un saut non local, comme un appel de fonction. Pour cette situation, on dispose également de deux instructions qui, avant de sauter, sauvegardent une adresse de retour dans le registre `$ra`, qui permettra plus tard de revenir à l'exécution de la séquence en cours.

```
jal  label
jalr $t0
```

On reviendra sur ce mécanisme d'appel de fonction dans un chapitre ultérieur.

Appels système. Pour certaines fonctionnalités dépendant du système d'exploitation, un code assembleur doit faire appel aux bibliothèques système. Dans ce cours, on exécutera notre code MIPS dans un simulateur, qui prendra en charge ces différents services avec une instruction dédiée. On décrit donc ici des fonctionnalités du simulateur, qui miment des services dépendant normalement du système d'exploitation.

On déclenche un appel système à l'aide de l'instruction `syscall`, après avoir placé dans le registre `$v0` un code désignant le service demandé. Voici une petite sélection :

service	code	arg.	rés.
affichage	1 (entier) 4 (chaîne) 11 (ascii)	`$a0`	
lecture	5 (entier)		`$v0`
arrêt	10		
extension mémoire	9	`$a0`	`$v0`

On trouve encore des services pour la manipulation de fichiers, et d'autres variantes d'affichage ou de lecture.

Hello, world. Le programme le plus célèbre de la terre, en MIPS.

```
        .text
main:   li   $v0, 4
        la   $a0, hw
        syscall
        li   $v0, 10
        syscall

        .data
hw:     .asciiz "hello world\n"
```

Notes : 4 est le code de l'appel système d'affichage d'une chaîne de caractères, 10 celui de l'arrêt. L'annotation `.asciiz` permet de placer une chaîne de caractères dans les données statiques, dont l'adresse est ici associée à l'étiquette `hw`.

4.3 Approfondissement : traduction complète pour IMP

Voyons comment traduire un programme IMP en un programme assembleur MIPS.

Module auxiliaire MIPS Pour représenter le code MIPS, on se donne un type minimal permettant de représenter efficacement des concaténations de chaînes de caractères.

```
type asm =
  | Nop
  | S of string
  | C of asm * asm

let (@@) x y = C (x, y)

type program = { text: asm; data: asm; }
```

Ainsi, pour « concaténer » deux fragments a_1 et a_2 de code assembleur, il suffit de former la structure `C(`a_1`, `a_2`)`, ce que l'on peut abréger avec la notation a_1 `@@` a_2. C'est plus efficace qu'une vraie concaténation, qui aurait un coût proportionnel à la longueur des chaînes manipulées.

Pour faciliter la production de code MIPS sous cette représentation, on définit en plus de cela quelques constantes pour les registres utilisés

```
let t0 = "$t0"
let t1 = "$t1"
let a0 = "$a0"
let v0 = "$v0"
let sp = "$sp"
let ra = "$ra"
```

ainsi qu'une série de fonctions auxiliaires produisant des instructions MIPS. L'objectif est que le code caml produisant une représentation de code assembleur MIPS soit aussi proche que possible du code MIPS représenté. On pourra ainsi écrire en caml dans du code caml l'expression `add t0 t0 t1` pour inclure la chaîne de caractères `"  add $t0, $t0, $t1\n"` dans un élément de type `asm`.

Voici les premières de ces fonctions.

```
open Printf
let li   r1 i      = S(sprintf "  li   %s, %i"     r1 i)
let la   r1 x      = S(sprintf "  la   %s, %s"     r1 x)
let move r1 r2     = S(sprintf "  move %s, %s"     r1 r2)

let add  r1 r2 r3  = S(sprintf "  add  %s, %s, %s" r1 r2 r3)
let addi r1 r2 i   = S(sprintf "  addi %s, %s, %d" r1 r2 i)
let mul  r1 r2 r3  = S(sprintf "  mul  %s, %s, %s" r1 r2 r3)
let slt  r1 r2 r3  = S(sprintf "  slt  %s, %s, %s" r1 r2 r3)
let and_ r1 r2 r3  = S(sprintf "  and  %s, %s, %s" r1 r2 r3)

let j    l         = S(sprintf "  j    %s"         l)
let jal  l         = S(sprintf "  jal  %s"         l)
let jr   r1        = S(sprintf "  jr   %s"         r1)
let beqz r1 l      = S(sprintf "  beqz %s, %s"     r1 l)
let bnez r1 l      = S(sprintf "  bnez %s, %s"     r1 l)
let bltz r1 l      = S(sprintf "  bltz %s, %s"     r1 l)
let bge  r1 r2 l   = S(sprintf "  bge  %s, %s, %s" r1 r2 l)

let syscall         = S("  syscall")
let nop             = Nop
let label l         = S(sprintf "%s:" l)
let comment s       = S(sprintf "  # %s" s)
```

Pour les instructions MIPS de la famille de `lw` et `sw`, on crée des fonctions qui prennent trois arguments, dans l'ordre le registre sur lequel on agit, le décalage, et le registre donnant l'adresse de base. On écrit donc `lw t0 4 t1` pour générare l'instruction `lw $t0, 4($t1)` qui charge dans `$t0` la valeur trouvée avec un décalage de `4` à partir de l'adresse donnée par `$t1`.

```
let lw   r1 o r2   = S(sprintf "  lw   %s, %d(%s)" r1 o r2)
let sw   r1 o r2   = S(sprintf "  sw   %s, %d(%s)" r1 o r2)
let lbu  r1 o r2   = S(sprintf "  lbu  %s, %d(%s)" r1 o r2)
```

Notez que la syntaxe caml autorise encore à écrire une version encore plus proche du format MIPS, puisque `lw t0 4(t1)` est bien l'application de `lw` aux trois arguments `t0`, `4` et `t1` (avec des parenthèses superflues autour de `t1`).

Pour traiter les déclarations de données statiques, on introduit encore une les fonctions auxiliaires suivantes.

```
let ilist = function
  | []     -> ""
  | [i]    -> sprintf "%i" i
  | i :: l -> sprintf "%i, %s" i (ilist l)
let dword l  = S(sprintf "  .word %s" (ilist l))
let asciiz s = S(sprintf "  .asciiz %s" s)
```

Cette batterie de fonctions auxiliaires donne une manière fluide de générer en caml une représentation de code MIPS. Par exemple, le code MIPS ci-dessous à gauche peut être défini par l'expression caml à droite.

```
  # built-in atoi
atoi:
  li    $v0, 0
  li    $t1, 10

atoi_loop:
  lbu   $t0, 0($a0)
  beqz  $t0, atoi_end
  addi  $t0, $t0, -48
  bltz  $t0, atoi_error
  bge   $t0, $t1 atoi_error
  mul   $v0, $v0, $t1
  add   $v0, $v0, $t0
  addi  $a0, $a0, 1
  j     atoi_loop

atoi_error:
  li    $v0, 10
  syscall

atoi_end:
  jr    $ra
```

```
let built_ins =
  comment "built-in atoi"
  @@ label "atoi"
  @@ li    v0 0
  @@ li    t1 10

  @@ label "atoi_loop"
  @@ lbu   t0 0(a0)
  @@ beqz  t0 "atoi_end"
  @@ addi  t0 t0 (-48)
  @@ bltz  t0 "atoi_error"
  @@ bge   t0 t1 "atoi_error"
  @@ mul   v0 v0 t1
  @@ add   v0 v0 t0
  @@ addi  a0 a0 1
  @@ j     "atoi_loop"

  @@ label "atoi_error"
  @@ li    v0 10
  @@ syscall

  @@ label "atoi_end"
  @@ jr    ra
```

Pour finir, on ajoute deux fonctions `push` et `pop` produisant des séquences de code MIPS facilitant la manipulation de la pile.

- `push` *r* génère du code MIPS qui empile le contenu du registre *r*, et
- `pop` *r* génère du code MIPS qui retire l'élément au sommet de la pile et le place dans le registre *r*.

Ces deux éléments n'existent pas directement dans les instructions ou pseudo-instructions MIPS. Via l'utilisation de notre module MIPS on pourra les voir comme des *pseudo*-pseudo-instructions.

```
let push r =
  addi sp sp (-4) @@ sw r 0(sp)
let pop r =
  lw r 0(sp) @@ addi sp sp 4
```

Cette représentation caml des programmes MIPS étant fixée, une simple fonction permet ensuite d'écrire l'ensemble d'un programme MIPS vers une destination cible, par exemple un fichier.

```
let rec print_asm fmt a =
  match a with
  | Nop        -> ()
  | S s        -> fprintf fmt "%s\n" s
  | C (a1, a2) ->
     let () = print_asm fmt a1 in
     print_asm fmt a2

let print_program fmt p =
  fprintf fmt ".text\n";
  print_asm fmt p.text;
  fprintf fmt ".data\n";
  print_asm fmt p.data
```

Traduction de IMP vers MIPS. Pour traduire les expressions de la manière la plus simple posisble, on va réaliser en MIPS quelque chose de semblable à la machine virtuelle à pile vue dans le chapitre d'introduction. Plus précisément :

- la fonction `tr_expr`, appliquée à une expression `e`, produit un code assembleur qui calcule la valeur de `e` et place le résultat dans le registre `$t0`,
- toutes les valeurs intermédiaires sont enregistrées sur la pile.

Pour manipuler la pile, on utilisera les deux fonctions `push` et `pop` définies dans notre bibliothèque MIPS.

La fonction de traduction d'une expression génère une séquence d'instructions assembleur MIPS plaçant la valeur calculée dans le registre `$t0`.

```
let rec tr_expr = function
  | Cst n -> li t0 n
  | Var x -> la t0 x @@ lw t0 0(t0)
```

Lorsque l'on traduit une opération binaire, on calcule d'abord la valeur de l'opérande de gauche. Puis on stocke cette valeur sur la pile le temps de calculer également la valeur de l'opérande de droite. À la fin, on récupère la première valeur sur la pile et on complète l'opération. On utilise au total deux registres : `$t0` qui recueille systématiquement la dernière valeur calculée, et `$t1` qui sert temporairement au moment de compléter une opération binaire.

```
  | Bop(bop, e1, e2) -> let op = match bop with
                          | Add -> add
                          | Mul -> mul
                          | Lt  -> slt
                          | And -> and_
                        in
                        tr_expr e1
                        @@ push t0
                        @@ tr_expr e2
                        @@ pop t1
                        @@ op t0 t1 t0
```

Pour traduire des boucles et des branchements, nous avons besoin d'introduire des sauts, et donc des étiquettes. On se donne donc une fonction auxiliaire `new_label` qui génère à chaque appel une nouvelle étiquette à l'aide d'un compteur interne.

```
let new_label =
  let cpt = ref (-1) in
  fun s -> incr cpt; Printf.sprintf "__%s_%d" s !cpt
```

La fonction de traduction d'une instruction génère également une séquence d'instructions MIPS.

```
let rec tr_instr = function
  | Print e ->
     tr_expr e
     @@ move a0 t0
     @@ li v0 11
     @@ syscall
  | Set(x, e) ->
     tr_expr e
     @@ la t1 x
     @@ sw t0 0(t1)
```

La traduction d'une boucle demande la création de plusieurs étiquettes : une pour chaque instruction susceptible d'être la cible d'un saut. En l'occurrence, on a deux instructions de saut et deux étiquettes cible :

- un saut conditionnel (`beqz`) pour sortir de la boucle si le test est négatif, en ciblant une étiquette `end_label` placée après le corps la boucle,
- un saut inconditionnel (`j`) pour revenir au test (étiquette `loop_label` après chaque exécution du corps de la boucle.

```
  | While(e, s) ->
     let loop_label = new_label "loop"
     and end_label = new_label "endloop"
     in
     label loop_label
     @@ tr_expr e              (* test *)
     @@ beqz t0 end_label
     @@ tr_seq s               (* loop body *)
     @@ j loop_label
     @@ label end_label
```

De même pour une instruction de branchement, avec un saut conditionnel allant vers la branche **else**, et (attention, piège !) un saut inconditionnel à la fin de la branche **then** vers une étiquette `end_label` pour *ne pas passer* par la branche **else**.

```
  | If(e, s1, s2) ->
     let else_label = new_label "else"
     and end_label = new_label "endif"
     in
     tr_expr e                (* test *)
     @@ beqz t0 else_label
     @@ tr_seq s1             (* then branch *)
     @@ j end_label
     @@ label else_label
     @@ tr_seq s2             (* else branch *)
     @@ label end_label
```

Enfin, la traduction d'une séquence d'instruction est simplement la combinaison des traductions des instructions individuelles.

```
and tr_seq = function
  | []   -> nop
  | [i]  -> tr_instr i
  | i::s -> tr_instr i @@ tr_seq s
```

Pour compléter ce programme de traduction, il nous faut encore détecter toutes les variables utilisées par le programme IMP pris en argument, afin de les déclarer dans notre segment des données statiques. Voici une fonction simple réalisant cette analyse.

```
module VSet = Set.Make(String)
let rec vars_expr = function
  | Cst _ -> VSet.empty
  | Var x -> VSet.singleton x
  | Bop(_, e1, e2) ->
     VSet.union (vars_expr e1) (vars_expr e2)
let rec vars_instr = function
  | Print e -> vars_expr e
  | Set(x, e) ->
     VSet.add x (vars_expr e)
  | While(e, s) ->
     VSet.union (vars_expr e) (vars_seq s)
  | If(e, s1, s2) ->
     VSet.union (vars_expr e)
                (VSet.union (vars_seq s1) (vars_seq s2))
and vars_seq = function
  | []   -> VSet.empty
  | i::s -> VSet.union (vars_instr i) (vars_seq s)
```

Pour traduire un programme IMP complet, on combine alors la traduction des instruction et l'allocation de ses variables globales.

```
let tr_prog p =
  let text = tr_seq p in
  let vars = vars_seq p in
  let data = VSet.fold
    (fun id code -> label id @@ word [0] @@ code)
    vars nop
  in
  { text; data }
```