Université Denis Diderot
Licence d'Informatique

Les Langages de Programmation

syntaxe, sémantique et implantation

Guy Cousineau

Janvier 2005

Table des matières

Introduction

Le but de ce cours est de prendre un peu de recul par rapport aux langages de programmation, de définirr des critères permettant de les comparer, de mettre en évidence les choix fondamentaux faits pour chacun d'eux et la cohérence interne qu'ils développent à partir de ces choix fondamentaux.
Les principaux langages considérés seront C, JAVA et CAML. Des références à d'autres langages comme PASCAL, C++, OBJECTIVE C et SCHEME pourront être faites ponctuellement.

Une façon traditionnelle de classifier ces langages consiste à dire que PASCAL et C sont des langages procéduraux, que C++, OBJECTIVE C et JAVA sont des langages à objets et que SCHEME et CAML sont des langages fonctionnels. Cependant, d'autres critères permettent des classifications différentes.

Par exemple, JAVA, SCHEME et CAML ont en commun la particularité de fonctionner avec un récupérateur automatique de mémoire (garbage collector). Cela leur permet d'utiliser implicitement des pointeurs sans que l'utilisateur ait à se préoccuper de gérer directement ces pointeurs. La récupération automatique de mémoire a de nombreuses conséquences sur la représentation des structures de données. Par exemple, la taille des tableaux doit être une information disponible dans leur représentation à l'exécution.

Si on se place du point de vue de la structure de blocs et en particulier de l'emboîtement des définitions de fonctions ou de procédures, alors on est amené à rapprocher PASCAL de SCHEME et CAML puisque ces langages sont les seuls à autoriser des définitions emboîtées qui nécessitent des méthodes spécifiques de traitement des variables libres dans les corps de fonctions (liens statiques, displays ou fermetures).

Enfin, si on se place du point de vue de la dynamique des programmes et des décisions qui peuvent être prises à l'exécution, alors on est amené à rapprocher SCHEME et les langages à objets comme JAVA ou OBJECTIVE C. SCHEME permet le test dynamique de type et JAVA et OBJECTIVE C le test dynamique d'appartenance à une classe.

De nombreux autres critères de comparaison peuvent être envisagés permettant d'autres rapprochements et d'autres distinctions. Ce qui est intéressant est précisément de pouvoir mettre en évidence les choix qui interviennent

dans la conception d'un langage et les conséquences de ces choix. Pour cela, il faut tout d'abord préciser comment on peut décrire un langage de programmation dans tous ses aspects syntaxiques et sémantiques. Ce sera l'objet du premier chapitre.

Chapitre 1

Diversité des langages

1.1 Les niveaux de syntaxe

1.1.1 La syntaxe concrête

Le premier niveau de description d'un langage de programmation est celui de sa syntaxe. Cette syntaxe se présente concrêtement sous forme de chaînes de caractères formant le texte d'un programme. Pour spécifier quels sont les programmes syntaxiquement corrects, on utilise des **grammaires** qui définissent les règles de formation de programmes.

La description de la syntaxe concrête d'un langage se décompose en fait en deux couches. On décrit d'abord la structure lexicale du langage puis sa structure syntaxique.

La structure lexicale décrit les **lexèmes** ou **unités lexicales** du langage c'est-à-dire la façon dont sont écrits les mots-clés, les identificateurs, les nombres, les opérateurs et autres symboles utilisés par le langage. Chacun de ces lexèmes est écrit avec un ou plusieurs caractères. Par exemple, les caractères `+` et `=` servant à représenter en C l'opérateur d'addition et l'opérateur d'affectation définissent, lorsqu'ils sont utilisés seuls deux lexèmes que l'on pourra par exemple appeler `PLUS` et `ASSIGN`. Par contre, utilisés ensemble sous la forme `+=`, ils correspondent à un autre lexème que l'on pourra par exemple appeler `PLUSASSIGN`. De même un nombre entier, un nombre flottant ou un identificateur correspondront chacun à un lexème (`CST_INT`, `CST_FLOAT` ou `IDENT`) bien qu'ils puissent être formés d'un nombre arbitraire de caractères.

L'analyse lexicale d'un langage consiste à regrouper de façon convenable en lexèmes la suite de caractères représentant un programme.

Par exemple, si nous utilisons la correspondance entre caractères et lexèmes donnée dans la table suivante,

```
+   PLUS              +=   PLUSASSIGN
*   STAR
(   LPAREN            )    RPAREN
[   LBRACKET          ]    RBRACKET
```

La suite de caractères

```
tab[index] += tan(*p+1)
```

produira la suite de lexèmes

```
IDENT("tab")  LBRACKET  IDENT("index")  RBRACKET PLUSASSIGN
IDENT("tan")   LPAREN STAR  IDENT("p")  PLUS CST_INT(1) RPAREN
```

Un fois cette analyse lexicale effectuée, l'analyse syntaxique extrait de la suite de lexèmes produite la structure syntaxique du programme qui est définie par une grammaire.
Par exemple, la structure syntaxique des expressions du langage C pourra être définie (partiellement) par la grammaire suivante, donnée ici dans la syntaxe de YACC. Les identificateurs en majuscules sont des lexèmes.

```
expression:
        constant
|       IDENT
|       STAR expression
|       LPAREN expression RPAREN
|       IDENT  LPAREN  RPAREN
|       IDENT  LPAREN arguments RPAREN
|       expression LBRACKET expression RBRACKET
|       expression PLUS expression
|       expression ASSIGN expression
|       expression PLUSASSIGN expression
|             ........
;

constant:
        CST_INT
|       CST_FLOAT
;
```

1.1.2 La syntaxe abstraite

La vérification du fait qu'un programme est syntaxiquement correct par rapport à la grammaire partielle donnée plus haut peut être réalisée par un programme appelé **analyseur syntaxique**. Un tel analyseur, cependant, ne se contente pas en général de vérifier la correction syntaxique du programme analysé : il produit aussi à partir de ce programme une représentation intermédiaire qui va être utilisée pour des traitements à effectuer sur le programme (typage, compilation, etc.). C'est cette représentatiom qu'on appelle **syntaxe abstraite** du programme.

Cette syntaxe abstraite peut être vue comme un terme ou un arbre qui met en évidence la structure du programme analysé et oublie volontairement tous les détails inutiles. Par exemple, en C, les corps de fonctions sont parenthésés par des accolades alors qu'en PASCAL, elles sont parenthésées par les mots-clés **begin** et **end**. Ceci est typiquement un détail de syntaxe concrête qui sera oublié au niveau de la syntaxe abstraite.
La syntaxe abstraite peut aussi regrouper en une même description des constructions concrêtes différentes qui ont le même sens. Par exemple, en C, on peut déclarer simultanément plusieurs identificateurs de types différents pourvu que leur type final soit le même :

```
int tab[] , *ptr, n;
```

mais on peut aussi déclarer indépendamment chacun de ces identificateurs :

```
int tab[];
int *ptr;
int n;
```

Il n'y a pas lieu de distinguer ces deux formes équivalentes au niveau de la syntaxe abstraite.

1.1.3 Description de la syntaxe abstraite

La description de la syntaxe abstraite d'un langage, se fait en définissant :

1. Un ensemble de **catégories syntaxiques** : dans un programme, on identifie typiquement des catégories comme **declaration**, **type**, **instruction**, **expression**, etc.
2. Pour chaque catégorie, un ensemble de **constructeurs** permettant de construire des objets de cette catégorie. Par exemple, dans la catégorie **instruction**, on aura un constructeur For corespondant aux boucles for, un constructeur While correspondant aux boucles while etc. Ces constructeurs ont en général des arguments. Par exemple, pour définir une boucle while, on a besoin d'un test, qui est une expression, et d'un corps de boucle, qui est une instruction. Le constructeur While aura donc deux arguments, une expression et une instruction.

Les constructeurs sont aussi utilisés pour rendre compte de certaines formes d'inclusions entre catégories et dans ce cas, ils peuvent ne pas avoir de contrepartie au niveau de la syntaxe concrête.
Par exemple, en C, une constante est une expression et une expression suivie d'un point-virgule est une instruction. Dans la syntaxe abstraite, un constructeur permettra de transformer une constante en expression et un autre constructeur permettra de transformer une expression en instruction. Le second correspondra au point-virgule de la syntaxe concrête alors que le premier n'aura pas de contrepartie dans la syntaxe concrête.

Voici comment on peut décrire la syntaxe abstraite d'un sous-ensemble de C :

CATEGORIES :
declaration, **type**, **instruction**, **expression**, **ident**, **binop**, **constante**

CONSTRUCTEURS :
declaration
- VarDecl(**type**,**ident**)
- FunDEcl(**type**,**ident**,**declaration list**,**instruction**)

type
- IntType
- FloatType
- ArrayType(**type**)
- PtrType(**type**)

instruction
- Block(**declaration list**,**instruction list**)
- Comput(**expression**)
- While(**expression**,**instruction**)
- For(**expression**,**expression**,**expression**,**instruction**)
- If(**expression**,**instruction**)
- Return(**expression**)

expression
- ConstExp(**constante**)
- VarExp(**ident**)
- BinopExp(**expression**,**binop**,**expression**)
- IndexExp(**expression**,**expression**)
- RefExp(**expression**)
- FunCallExp(**expressionlist**)

binop
- Assign
- Add
- Mult
- Less

constante
- Int(**int**)
- Float(**float**)

Les constructeurs permettent de représenter la syntaxe abstraite d'un programme sous la forme d'un terme. Par exemple, le programme

```
int fact(int n)
{
 int i, m;
 m=1;
 for (i=2;i<n+1;i=i+1)    m=m*i;
```

```
 return(m);
   }
```

a pour syntaxe abstraite :

```
FunDecl(IntType, "fact", [VarDecl(IntType,"n")],
        Block([VarDecl(IntType,"i");[VarDecl(IntType,"m")],
              [Comput(BinopExp("m", Assign, Int 1));
               For(BinopExp("i", Assign, Int 2),
                   BinopExp("i", Less, BinopExp("n", Add, Int 1)),
                   BinopExp("i", Assign, BinopExp("i", Add, Int 1)),
                   Comput(BinopExp("m", Assign, BinopExp("m", Mult, "i") )));
               Return(Varexp "m")]))
```

En reprenant, notre exemple précédent, la suite de caractères

```
tab[index] += tan(*p+1)
```

transformée en la suite de lexèmes

```
IDENT("tab")  LBRACKET  IDENT("index")  RBRACKET PLUSASSIGN IDENT("tan")
LPAREN STAR  IDENT("p")  PLUS CST_INT(1) RPAREN
```

produira après analyse syntaxique

```
BinopExp(IndexExp(VarExp "tab", VarExp "index"),
         PlusAssign,
         FunCallExp("tan",[BinopExp(RefExp(VarExp "p"),Add,Int(1))]))
```

Les termes de syntaxe abstraite peuvent être visualisés comme des arbres qui mettent en évidence leur structure. Par exemple, le terme précédent est dessiné à la figure 1.1 :

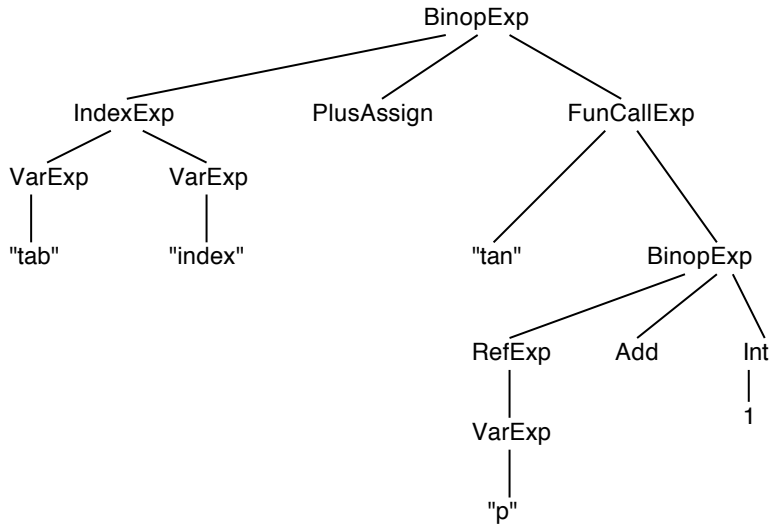


FIG. 1.1 – Un arbre de syntaxe abstraite

A partir d'un terme de syntaxe abstraite, il est très facile de reconstituer une syntaxe concrête[1] mais l'opération inverse est beaucoup plus difficile puisqu'elle nécessite une analyse syntaxique.

1.1.4 Les générateurs d'analyseurs syntaxiques

L'écriture d'un analyseur syntaxique pour un langage de programmation est une tâche relativement difficile même en utilisant des outils de génération automatique. Par exemple, l'utilisation de YACC demande que la grammaire que l'on fournit pour le langage ait une propriété assez forte (celle d'être LALR(1), voir [2]). Par contre, il est en général plus facile d'augmenter ou de modifier une grammaire existante.
Un analyseur syntaxique doit non seulement être capable de vérifier la correction syntaxique d'un programme mais aussi construire le terme de syntaxe abstraite correspondant. Pour cela, la grammaire qu'on lui fournit doit associer à chaque règle ce qu'on appelle une **action sémantique**, c'est-à-dire une expression fabriquant la syntaxe abstraite correspondant à une règle en fonction des syntaxes abstraites associées à ses constituants. Dans la tradition de YACC, on note par des variables `$i` la syntaxe abstraite associée au ième composant du membre droit d'une règle. La grammaire donnée plus haut peut être complétée d'actions sémantiques de la façon suivante. (Les actions sémantiques sont écrites entre accolades).

```
expression:
   constant
     {$1}
|  IDENT
     {$1}
|  STAR expression
     {RefExp($2)}
|  LPAREN expression RPAREN
     {$2}
|  IDENT  LPAREN  RPAREN
     {FunCallExp($1,[])}
|  IDENT  LPAREN arguments RPAREN
     {FunCallExp($1,$3)}
|  expression LBRACKET expression RBRACKET
     {IndexExp($1,$3)}
|  expression PLUS expression
     {BinopExp($1,Add,$3)}
|  expression ASSIGN expression
     {BinopExp($1,Assign,$3)}
|  expression PLUSASSIGN expression
     {BinopExp($1,PlusAssign,$3)}
|               ........
;
```

[1] qui n'est pas nécessairement celle d'origine mais qui possède la même sémantique

```
constant:
   CST_INT
     {Int $1}
|  CST_FLOAT
     {Float $1}
;
```

Les principes des analyseurs LALR sont décrits dans le chapitre 2 et leur utilisation pratique dans le chapitre 3.

1.2 Portée des identificateurs et environnements

Dans tout langage de programmation, il existe différentes constructions permettant d'introduire des variables. Ce sont par exemple la définition de fonctions et l'ouverture d'un bloc. On appelle de telles constructions des **lieurs** de variables.
Le sens des variables qui figurent dans les expressions ou les instructions d'un programme doit toujours être interprété par rapport à leur introduction c'est-à–dire que l'on doit toujours être capable de rattacher une utilisation de variable à son lieur pour pouvoir l'interpréter correctement.
Ce n'est pas toujours évident car plusieurs variables d'un programme peuvent avoir le même nom. Un même identificateur peut être utilisé par exemple pour un paramètre de fonction et pour une variable locale déclarée dans un block à l'intérieur du corps de la fonction, et ceci éventuellement avec des types différents.

La plupart des langages de programmation utilisent la même règle pour relier une occurence de variable à son lieur et cette règle est très facile à expliquer sur l'arbre de syntaxe abstraite. Pour rattacher une utilisation de variable à son lieur, il suffit de remonter la branche de l'arbre sur laquelle se trouve l'occurence de variable jusqu'à trouver un lieur qui définit cette variable. Le premier trouvé est le bon. En d'autres termes, c'est le lieur le plus proche, c'est-à-dire celui qui a introduit en dernier la variable, qui est à utiliser. La portée d'un lieur est le sous-arbre dont il est la racine sauf que, à l'intérieur de ce sous-arbre peuvent apparaître d'autres lieurs cachant certaines liaisons que le premier a introduites.

Pour gérer cette relation entre une utilisation de variable et son introduction, nous utiliserons la notion d'environnement. Un environnement peut être vu comme une table qui associe à toute variable introduite une ou plusieurs propriétés, par exemple, son type ou bien sa valeur. Le fait de relier une utilisation de variable à sa définition la plus proche permet de gérer les environnement comme des piles. Nous expliquerons cette gestion en considérant pour le moment des environnements de typage, c'est-à-dire des environnement où les variables sont associées à leur type.

– Pour construire l'environnement de typage valable en un point du programme, on parcourt la branche qui va de la racine de l'arbre de syntaxe abstraite à ce point. On part avec un environnement vide et chaque fois qu'on traverse un lieur, on empile les liaisons introduites par ce lieur.
– Pour connaitre le type d'une occurence de variable, on prend l'environnement de typage correspondant à cette occurence et on le parcourt depuis le sommet de pile jusqu'à trouver la première liaison de la variable. Cette liaison indique le bon type. Si on ne trouve pas de liaison dans l'environnement, c'est que la variable est utilisée sans avoir été introduite, ce qui est une erreur.

Par exemple, dans le programme PASCAL suivant

```
function f(m:integer, n:integer): integer;
begin
  function g(n:boolean,p:integer):integer;
       begin if n then g:=m+p else g:=m*p end
  m:integer;
  m:=7;
  f:= g(true,m+n)
end
```

l'environnement correspondant à l'instruction
`if n then g:=m+p else g:=m*p`
est
`[p:integer;n:boolean; n:integer; m:integer]`
(en plaçant le haut de pile à gauche)
et l'environnement correspondant à l'instruction
`f:= g(true,m+n)`
est `[m:integer; n:integer; m:integer]`.

Le problème de portée des identificateurs est assez simple en ce qui concerne le typage mais il peut être nettement plus complexe quand on considère l'exécution des programmes. Par exemple, l'exécution de l'appel de fonction `f(2,3)` où la fonction `f` est celle définie plus haut rend la valeur 12 et non pas, comme on pourrait le croire, la valeur 17. En effet, bien que la valeur de la variable `m` courante soit 7 au moment de l'appel `g(true, m+n)`, ce qui revient donc à évaluer `g(true,10)`, la variable `m` utilisée dans le corps de la fonction `g` vaut 2 car cette variable `m` est le paramètre `m` dans la fonction `f` et non pas la variable `m` qui a été introduite dans le corps de la fonction `f` postérieurement à la définition de `g`.
Si on y réfléchit, cette interprétation de la variable `m` dans le corps de `g` est la seule qui soit raisonnable car la nouvelle variable `m` introduite dans le corps de `f` aurait parfaitement pu avoir un type différent de `integer`. Si on veut que le typage garantisse l'absence d'erreur de type à l'exécution, il est indispensable que l'exécution ne remette pas en cause la validité des environnements utilisés pour le typage. Cete définition de la portée des identificateurs est

connue sous le nom de **portée statique**. Elle est utilisée dans la plupart des langages sauf dans certaines versions anciennes de LISP et aussi, pour ce qui est des méthodes, dans les langages à objects.
La portée statique des identificateurs ne pose pas de problème d'implantation dans les langages qui n'autorisent pas l'emboitement des définitions de fonctions comme C et ses dérivés ou JAVA. Par contre, pour les autres elle implique soit une compilation délicate des accès au variables (par exemple en PASCAL avec la notion de **display**) soit l'utilisation de **fermetures** pour représenter les fonctions comme en CAML ou en SCHEME.

1.3 Les types

1.3.1 Typage statique et typage dynamique

Le système de types d'un langage de programmation reflète la nature des structures de données fournies par ce langage et permet un contrôle sur l'utilisation correcte de ces données. Ce contrôle est surtout utile s'il s'exerce à la compilation c'est-à-dire avant l'exécution des programmes. Il permet alors de détecter un certain nombre d'erreurs de programmation qui, sans cela, conduiraient à des erreurs d'exécutions. Par ailleurs, lorsque le typage est vérifié à la compilation (typage statique), le code compilé n'a pas besoin de contenir de vérifications dynamique de types et il est donc potentiellement plus efficace.
Cependant, le typage statique introduit aussi des contraintes. Certains programmes corrects peuvent être refusés par le vérificateur de types alors qu'ils ne produiraient pas d'erreurs à l'exécution. Certains programmes peuvent avoir des comportements dynamiques plus intéressants si certaines décisions sont prises à l'exécution et non pas à la compilation. En particulier pour les langages à objets, l'interprétation dynamique plutôt que statique des messages est un élément de souplesse important.
Dans cette section, on s'intéressera uniquement au typage statique. Les aspects dynamiques seront abordés dans la section 1.5.

1.3.2 Les garanties apportées par le typage statique

On dira qu'un langage de programmation est **fortement typé** si l'acceptation d'un programme par le compilateur garantit l'absence d'erreurs de types à l'exécution.
C'est loin d'être le cas pour tous les langages typés. Il s'agit d'une proprété idéale dont les langages réels sont souvent fort éloignés. En particulier le langage C, qui autorise de nombreuses conversions de types hasardeuses, est loin de satisfaire ce critère. Une victime célèbre de ces conversions hasardeusse est la fusée Ariane 5 dont le premier vol a échoué à cause d'une conversion sur 16 bits d'un entier 32 bits.

Les autres langages offrent plus de garanties. En particulier C++ apporte beaucoup plus de garanties que C dans ce domaine. Une expérience intéressante consiste à prendre un logiciel écrit en C et à le recompiler avec le compilateur C++. On détecte alors en général des conversions implicites que C++ n'accepte pas à juste raison. Cependant, le bénéfice apporté par C++ est en partie perdu si le programme force les conversions par des casts explicites.
Les langages CAML et JAVA se rapprochent beaucoup plus de langages fortement typés mais ils ont aussi certaines failles. Le typage de CAML ne prend pas en compte les exceptions et donc, un programme correctement typé peut engendrer une exception non rattrapée sans que cela se voie au typage. Par ailleurs, ces langages autorisent par souci d'interopérabilité l'utilisation de modules externes écrits en assembleur ou en C qui n'offrent pas de garanties de typage. Au moins, ces langages offrent-ils un noyau fortement typé.

En pratique, pour effectuer des vérifications qui ne sont pas faites au typage, on peut ajouter une phase d'analyse statique qui vérifie des propriétés importantes comme le respects des bornes des tableaux et la correction des conversions de types.

1.3.3 Les types prédéfinis

Dans tous les langages, il existe des types prédéfinis correspondant à différentes sortes de nombres et en général aux valeurs booléennes, aux caractères et aux chaînes de caractères. A ces types sont associées des constantes telles que 13, 3.14, `true`, `'a'`, `"hello"` qui sont utilisables directement dans les programmes.

1.3.4 Les constructeurs de types prédéfinis

Il existe aussi des constructeurs de types tels que pointeur (pointer) ou tableau (array). Ces constructeurs ne sont pas directement des types en eux-même mais ils permettent de construire des types quand on les applique à un type de base (ou à plusieurs types de base).
La façon dont on note les types construits à l'aide de constructeurs de types varie suivant les langages. En PASCAL, le type "pointeur sur entier" se note `(^integer)` et en CAML `(int ref)`. En C, il n'y a pas véritablement de dénotation pour un tel type mais plutôt une syntaxe pour déclarer une variable `p` de ce type par
`int *p;`.

Cependant, au niveau de la syntaxe abstraite, il est nécessaire d'avoir une dénotation pour chaque type. C'est la raison pour laquelle, nous avons introduit dans notre syntaxe abstraite de C, le constructeur `PtrType` à un argument, ce qui permet de noter `PtrType(IntType)` le type "pointeur sur entier".

La déclaration d'une variable de type tableau d'entier peut aussi prendre des formes assez variées selon le langage utilisé.
En PASCAL, les bornes du tableau, et par conséquent, sa taille, font partie du type, même lorsqu'il s'agit du type d'un paramètre de procédure ou de fonction :

```
tab: array 1..100 of integer;
function sumarray(t: array 1.100 of integer):integer;
```

En C et dans ses dérivés, la taille d'un tableau est nécessairement indiquée lorsque le tableau est alloué statiquement. Par contre, le tableau est passé en paramètre, cette information n'est pas exigée et elle n'est pas disponible au cours des calculs. On est alors amené à utiliser un deuxième paramètre pour passer la taille du tableau :

```
int tab[100];
int sumarray(int t[], int size);
```

En JAVA et en CAML, la taille du tableau ne fait pas partie du type mais elle intervient lorsqu'on construit (alloue) le tableau et elle est disponible au cours des calculs.
Il est donc naturel de considérer dans le cas de C, JAVA ou CAML, que tableau est un constructeur à un argument comme nous l'avons fait plus haut dans la syntaxe abstraite de C avec le constructeur `ArrayType`. Pour la syntaxe abstraite de PASCAL, on devrait plutôt considérer que le constructeur `ArrayType` a deux arguments, un pour le type des éléments du tableau et un autre pour le type intervalle qui définit ses bornes.

En fait, ce qu'il est important de retenir, c'est que les constructeurs de type fournissent le moyen d'introduire dans un langage de programmation une infinité de types qui sont obtenus en appliquant les constructeurs de type aux types de base et aussi aux types qui sont eux-mêmes construits de cette façon. Des exemples de tels types sont par exemple `array(int)` , `pointer(float)` ou encore `array(pointer(string))`
et `pointer(pointer(bool))`.

1.3.5 Les définitions de nouveaux types

Les langages traditionnels comme C ou PASCAL offrent essentiellement deux catégories de constructions pour introduire de nouveaux types. La premìere correspond à ce qu'on appelle le produit cartésien en mathématique et consiste à permettre de construire des n-uplets de valeurs c'est-à-dire de réunir dans un même objet plusieurs valeurs qui peuvent être de types différents. Ces constructions s'appellent des structures en C et des records (en français enregistements) en PASCAL. Une structure ou un enregistrement possèdent des champs qui contiennent des valeurs.

En PASCAL :
```
type point = record x:real; y:real end;
```
En C :
```
struct point {float x; float y;}
```

La deuxième catégorie de constructions correspond à ce qu'on appelle des sommes ou des unions disjointes en mathématique et consiste à réunir dans un même type plusieurs types préexistants. Ces constructions s'appelle des unions en C et des variantes en PASCAL. Les variantes de PASCAL sont restreintes en ce sens que les variantes portent uniquement sur certains champs d'un record et ne constituent pas une construction indépendante.
Il existe aussi en général une construction pour introduire des types à domaine fini (énumérations).

Une fois définis, ces types viennent compléter la liste des types prédéfinis du langage et ils s'utilisent de la même façon que ceux-ci dans les déclarations. Par ailleurs, ils peuvent être utilisés comme arguments des constructeurs de types.

1.3.6 Les définitions de nouveaux constructeurs de types

Outre la définition de nouveaux types, un langage comme CAML permet de définir aussi de nouveaux constructeurs de types. En fait, en CAML, on peut utiliser une seule et même construction pour les unions, les énumérations et la définitions de nouveaux constructeurs.
Voici la définition d'une énumération :

```
type couleur = Trefle | Carreau | Coeur | Pique;;
```

Les valeurs du type énuméré `couleur` sont `Trefle`, `Carreau`, `Coeur` et `Pique`.
Voici la définition d'une union :

```
type num = Int of int | Float of float;;
```

`Int` et `Float` sont les constructeurs du type `num` qui est l'union du type `int` et du type `float`. Il s'agit de constructeurs de valeurs. Par exemple `Int(3)` et `Float(2.5)` sont deux valeurs du type `num` obtenues en appliquant respectivement le constructeur `Int` à la valeur entière 3 et le constructeur `Float` à la valeur flottante 2.5. Notons que `Trefle`, `Carreau`, `Coeur` et `Pique` sont également des constructeurs de valeurs mais ce sont des constructeurs à 0 argument c'est-à-dire des constructeurs constants.

Les types sommes de CAML tirent leur principal intérêt du fait qu'ils peuvent être récursifs :

```
type arbre = F of int | Bin of arbre * arbre;;
```

On définit ainsi les arbres binaires ayant des entiers aux feuilles.

Pour définir un constructeur de type, il suffit de donner un ou plusieurs paramètres aux types définis. Par exemple, on peut redéfinir le type `arbre` de façon à ce qu'il soit paramétré par le type des feuilles :

```
type 'a arbre = F of 'a | Bin of 'a arbre * 'a arbre;;
```

Ici, `arbre` est un nouveau constructeur de type qui permet d'utiliser des types tels que `int arbre` ou bien `string arbre` ou encore `int array arbre`.

1.3.7 Le polymorphisme

On appelle **polymorphisme** le fait que certaines valeurs d'un langage de programmation peuvent avoir plusieurs types. Le polymorphisme peut se manifester de manière très diverse.
Tout d'abord, certaines valeurs spéciales ont naturellement plusieurs types. Par exemple, la valeur `nil` (ou `null`) qui désigne souvent un pointeur vide (un pointeur qui ne pointe sur rien) possède à la fois tous les types pointeurs.
De façon plus intéressante, certaines valeurs possèdent plusieurs types parce qu'elles appartiennent à un type qui est naturellement inclu dans un autre type. Par exemple, la valeur 3 possède le type entier mais dans l'expression $3 + 2.5$, elle possède le type flottant car le compilateur sait la transformer en un flottant avant de réaliser l'addition. Ce type de polymorphisme, qui apparait de façon un peu ad hoc dans la plupart des langages de programmation se trouve systématisé de façon plus rigoureuse dans les langages à objets. Si `B` est une sous-classe de `A`, alors tout objet de classe `B` peut aussi être considé comme un objet de classe `A`. Nous reviendrons sur ce point qui mérite un examen détaillé dans la section 1.3.9.

Le polymorphisme peut aussi se manifester au niveau des fonctions ou bien des méthodes des langages à objets et c'est à ce niveau-là qu'il est le plus intéressant. Une fonction (ou une procédure, ou une méthode) sera dite polymorphe si elle peut être appliquée (ou envoyée s'agissant de méthodes) à des objets de types différents. L'intérêt de ce polymorphisme tient au fait qu'il donne aux programmes utilisant cette fonction ou cette méthode des possibilités d'utilisation plus large puisqu'il peut s'appliquer à différents types de données. Considérons par exemple une procédure qui inverse l'ordre des éléments d'un tableau. Il s'agit d'une procédure qui peut être appliquée a priori à tous les tableaux, quelle que soit la nature des éléments du tableau. Cependant, dans un langage typé non polymorphe, il est impossible d'écrire une telle procédure car on doit indiquer le type du tableau passé en paramètre. Une telle procédure est un exemple typique de ce qu'on appelle le **polymorphisme paramétrique** dans lequel un même code peut être appliquée à des données de types différents. Cette forme de polymorphisme est à distinguer de la **surcharge** qui consiste à donner un même nom à des procédures différentes qui réalisent des des traitements plus ou

moins similaires sur des données différentes. Les langages à objets offrent une systématisation de cette notion de surcharge.

Nous examinerons séparément ces deux formes de polymorphisne.

1.3.8 Le polymorphisme paramétrique

Le polymorphisme paramétrique suppose que le langage de type du langage considéré soit suffisamment riche pour exprimer le fait que les types des arguments d'une fonction n'ont pas besoin d'être complêment spécifiés. Pour cela, on utilise des variables de type. Par exemple, en CAML, la fonction qui prend en argument un tableau et rend en résultat le tableau dans lequel tout les éléments ont été permutés peut s'écrire :

```
let permut t =
  let n = Array.length t - 1 in
  for i=0 to n/2
    do let x=t.(i)
       in t.(i)<-t.(n-i); t.(n-i)<-x
    done;
  t;;
```

et son type est

```
'a array -> 'a array
```

où `'a` est une variable de type. On peut appliquer cette fonction à un tableau d'entiers :

```
let tt = [|2;5;7|] in  permut tt;;
- : int array = [|7; 5; 2|]
```

ou à n'importe quel autre type de tableau. La fonction `permut` n'a pas besoin de connaitre le type des éléments du tableau pour effectuer la permutation.

1.3.9 Le polymorphisme des langages à objets

Le polymorphisme des langages à objets est de nature complètement différente. Un même méthode peut être définie dans de nombreuses classes, y compris dans des classes qui ne sont pas reliées entre elles hiérarchiquement. Par exemple, une méthode `display` peut être fournie dans toutes les classes qui définissent des objets visualisables sur l'écran. Ces différentes méthodes `display` peuvent avoir des codes complètement différents. On peut donc dire que la méthode `display` est surchargée de la même façon qu'un opérateur comme `+` peut être surchargé et désigner à la fois comme par exemple en JAVA l'addition des nombres et la concaténation des chaînes de caractères. Cependant, contrairement à l'opérateur `+` dont la surcharge doit être résolue de façon centralisée par le compilateur en tenant compte du type des arguments, la résolution de la surcharge de la méthode display peut être faite de façon décentralisée.

En fait, la façon dont cette surcharge est résolue dépend du langage considéré. En C++, la surcharge est résolue essentiellement par le compilateur de façon centralisée en fonction de la classe de l'objet à qui cette méthode est envoyée. La seule exception est celle des méthode déclarée "`virtual`". Si une méthode `m` est déclarée "`virtual`" dans une classe `A` et redéfinie dans une sous-classe `B` de `A`, alors si une variable `x` est typé statiquement comme étant de classe `A` mais se trouve à l'exécution être de classe `B`, un envoi de méthode `x.m()` fait appel à la méthode définie dans `B`. On a donc une forme de polymorphisme. Si on a construit un tableau d'objets de classe `A` mais que certains de ces objets sont réellement de classe `A` alors que d'autres sont de classe `B`, si on envoie la méthode `m` à tous les éléments du tableau, alors suivant le cas, c'est soit la méthode `m` de `A` soit la méthode `m` de `B` qui sera exécutée. Ce comportement est aussi celui de JAVA.

Dans d'autres langages comme OBJECTIVE C, il est possible de manipuler des objets dont on ne sais rien statiquement. Pour cela, on déclare que ces objets sont de type `id` qui désigne un pointeur sur un objet de classe quelconque. Si on construit un tableau d'objets de type `id` et qu'on envoie une méthode `m` à chacun des éléments du tableau, la méthode `m` utilisée est celle de chacun des objets, si elle existe. On a donc un polymorphisme plus large qu'en C++ ou en JAVA mais avec le risque qu'un objet ne sache pas répondre à la méthode `m`. On a donc un peu triché en contournant le système de type en introduisant le risque d'une erreur de type à l'exécution. OBJECTIVE C permet de prévenir ce type d'erreur en donnant la possibilité d'interroger les objets sur les méthodes qu'ils connaîssent de la même façon qu'en SCHEME par exemple, on peut tester le type d'une valeur à l'exécution. On utilise simplement ici le fait qu'un langage qui n'est pas typé statiquement est potentiellement polymorphe mais il s'agit d'un polymorphisme risqué.

1.3.10 Types abstraits, modules et classes

Outre leurs systèmes de types, les langages de progammation fournissent aussi des constructions permettant d'organiser les programmes à une échelle plus grande. Le développement de logiciels complexes nécessite en effet des outils pour découper ces logiciels en unités logiques, manipulables séparément et dont les interactions sont explicitées. Dans les langages traditionnels, ce découpage se fait essentiellement au niveau du système de fichiers. Par exemple, un logiciel écrit en C est découpé en fichiers qui peuvent être compilés séparément puis liés entre eux et à des bibliothèques pour fabriquer un exécutable.
Toutefois, un tel découpage en fichiers est assez rudimentaire car il n'offre pas de mécanisme permettant de spécifier la contribution que chaque fichier doit apporter à la réalisation du logiciel final et de séparer cette contribution de l'ensemble des structures de données et des fonctions et procédures qui

sont définies localement dans le fichier mais qui n'ont aucune raison d'être visibles à l'extérieur.
En d'autres termes, le découpage d'un logiciel en unités séparées n'est utile que si ces unités correspondent à une fonctionnalité bien identifiée : l'unité doit correspondre à un ensemble de tâches à réaliser. La façon dont l'unité s'y prend pour réaliser ces tâches ne devrait pas intervenir dans la spécification de l'unité et ne devrait pas être visible à l'extérieur de cette unité. En effet, la façon dont une unité fonctionnelle est réalisée peut être remise en cause au cours du développement. Il est donc important que les autres unités n'utilisent pas des détails d'implémentation qui peuvent être remis en cause. Sinon, elles devraient elles-mêmes être modifiées ultérieurement.
Pour résumer, une unité logicielle correspond à un **contrat** que ses développeurs s'engagent à respecter. Ils s'engagent sur ce contrat mais sur rien de plus. Quelle forme peut prendre ce contrat ? Supposons par exemple que l'unité est chargé d'implémenter une arithmétique sur des nombres entiers de grande taille. Son contrat consistera à fournir un type `bigint`, des opérations comme l'addition, la soutraction, la multiplication et la division sur ce type bigint, et des fonctions permettant des correspondances entre le type bigint et les types entier et flottant . Par exemple, on devra pouvoir construire un objet de type bigint à partir d'un entier ordinaire et obtenir un flottant approximant un bigint. Ceci peut se résumer par :

```
type bigint;
add_big: bigint -> bigint;
sub_big: bigint -> bigint;
mul_big: bigint -> bigint;
div_big: bigint -> bigint;
bigint_of_int: int -> bigint;
float_of_bigint: bigint -> float;
```

Cette table est bien sûr seulement une approximation du contrat du type bigint car elle passe sous silence l'essentiel à savoir que les fonctions fournies doivent implémenter une arithmétique correcte. Mais elle a le mérite de décrire ce qu'un programmeur pourra utiliser de cette unité. On remarque en particulier que la structure du type bigint n'est pas décrite. Le programmeur qui implémente cette unité aura le choix d'utiliser par exemple des tableaux d'entiers ou des listes chaînées. Ce choix sera transparent pour les utilisateurs de l'unité.

On appelle **type abstrait** la donnée d'un type dont l'implémentation n'est pas spécifiée et d'un ensemble de fonctions utilisant ce type. Un langage de programmation doit fournir des constructions pour représenter des types abstraits. Dans certains langages (ADA, CAML), ces constructions s'appellent des **modules**, dans d'autres comme TURBO PASCAL) des **unités**. Dans les langages à objets, c'est la notion de classe qui sert à la représentation des types abstraits. Ces constructions sont plus complexes que de simples types

abstraits car un module peut contenir de nombreux types (abstraits ou non), des exceptions, des sous-modules etc. Mais dans tous les cas, cette approche conduit à découper la présentation des unités fonctionnelles d'un logiciel en deux parties.
– Une partie publique qu'on appelle une **interface** et qui décrit ce qui sera visible à l'extérieur.
– Une partie non publique qui est l'**implémentation** de l'interface
Ce découpage assure que les unités qui compose un logiciel n'auront pas d'autres dépendances que celles décrites par les interfaces. Elle permet, par l'utilisation des interfaces, la compilation séparée des unités et leur réutilisation éventuelle dans d'autres logiciels sans qu'il soit nécessaire pour cela de rendre publique l'implémentation.

On notera cependant que les modules et les classes sont des outils de structuration extrêmement différents qui ont chacun leur logique. Les classes des langages à objets apportent essentiellement la notion d'héritage qui permet une organisation hiérarchique du code, utile dans de nombreuses applications. Par contre, elle offre peu de possibilité de paramétrisation. On retrouve ici la distinction entre le polymorphisme paramétrique et le polymorphisme d'inclusion.

1.4 L'évaluation

L'évaluation (ou exécution) des programmes peut utiliser des techniques assez variées qui font appel à deux notions principales : la notion d'interprétation et la notion de compilation.

1.4.1 La compilation

La notion de compilation est apparue en informatique à la fin des années 50 avec le langage FORTRAN. Avant FORTRAN, on programmait les ordinateurs en utilisant le jeu d'instructions fourni par l'unité centrale de l'ordinateur utilisé. Un programme était une suite d'instructions exécutables par la machine. Chaque instruction était inscrite sur une carte perforée et l'exécution était obtenue en copiant dans la mémoire de l'ordinateur le contenu de ces cartes à une adresse convenue puis en lançant l'exécution. Cette façon de faire avait deux gros inconvénients :
– Les programmes ne pouvaient tourner que sur un seul type de machine.
– Le programmeur devait coder lui-même son programme dans le jeu d'instructions très restreint qui lui était fourni.

FORTRAN (dont le nom signifie FORmula TRANslator) permettait au contraire de programmer dans un langage considéré à l'époque comme évolué : on pouvait utiliser des identificateurs symboliques (des variables) au lieu d'utiliser directement des adresses, des expressions arithmétiques et des

boucles for ! Un programme appelé compilateur permettait de traduire les programmes FORTRAN en instructions machines. Le programmeur pouvait donc, et c'était entièrement nouveau, écrire des programmes pouvant tourner sur plusieurs machines et le travail de codage était pris en charge mécaniquement. Cette notion de compilateur est une des grandes inventions de l'histoire de l'informatique et elle reste la technique de base permettant l'exécution des programmes. En particulier, pour un langage comme C qui est utilisé pour la programmation de couches logicielles fondamentales (le système UNIX notamment), c'est la technique d'implantation standard.

1.4.2 La compilation vers une machine virtuelle

Toutefois, à partir du moment où on dispose de compilateurs pour des langages flexibles et évolués comme C, on peut s'appuyer sur cet existant pour développer d'autres techniques d'implantation. On peut par exemple compiler vers C au lieu de compiler directement vers un langage machine. On peut ainsi se dispenser d'écrire un compilateur pour chaque type de processeurs : ce sont les compilateurs C qui se chargent de la production de code final pour chaque processeur. On peut aussi, et c'est une technique très utilisée aujourd'hui, définir ce que l'on appelle des machines abstraites (ou machines virtuelles). Ces machines abstraites peuvent avoir des jeux d'instructions beaucoup plus simples et beaucoup plus puissants que les processeurs réels et aussi mieux adaptés à la compilation des langages modernes. La compilation s'en trouve simplifiée, moins sujette aux erreurs et, ici encore, indépendante des processeurs réels. Il suffit ensuite d'écrire un programme C qui simule l'exécution de la machine abstraite pour disposer d'un mécanisme d'évaluation complet. C'est la technique d'implantation choisie par exemple par JAVA. Le code virtuel produit par le compilateur JAVA est portable, susceptible de vérification automatique élaborée, et c'est pourtant un code compilé à partir duquel il est difficile de reconstituer le code source : le secret du code source peut donc être préservé, ce qui peut être important dans un contexte industriel.

1.4.3 L'interprétation

On peut aussi envisager d'évaluer des programmes sans les compiler en écrivant directement un interpréteur du langage source dans un langage que l'on sait compiler. Cette technique revient à considérer le langage source comme le langage d'une machine virtuelle que l'on implante directement. Elle est peu utilisée en pratique car elle est assez inefficace. En effet, elle oblige l'interpréteur à parcourir sans arrêt l'arbre de syntaxe abstraite du programme. Il faut plutôt considérer les interpréteurs soit comme des implantations test réalisées pour disposer rapidement d'une implantation, par exemple dans la phase de conception d'un langage, soit comme des spécifi-

cations (exécutables) de la sémantique du langage c'est-à-dire comme des définitions du comportement que les programmes doivent avoir, comportement que le code produit par un compilateur devra imiter.

1.4.4 Les différents aspects de l'évaluation

Les deux aspects principaux de l'évaluation sont le contrôle et la gestion des données. Le contrôle définit quelles opérations doivent être réalisées et dans quel ordre. La gestion des données définit où sont stockées les données utilisées ou créées par le programme et comment ces données sont reliées aux noms de variables utilisées dans le programme source.

Dans un interpréteur, le contrôle est définit par l'arbre de syntaxe du programme source que l'interpréteur parcourt et les données sont contenues dans des variables de l'interpréteur. L'essentiel du travail de gestion des données est donc laissée à l'implantation du langage dans lequel est écrit l'interpréteur. Toutefois l'interpréteur doit gérer la correspondance entre les variables du programme source et les données correspondantes, ce qui est un aspect important des implantations. Pour cela, un interpréteur, va utiliser des environnements, c'est-à-dire, des tables de correspondances entre noms et valeurs et devra augmenter ces tables lorsqu'il rencontre des lieurs de variables dans la syntaxe du langage interprété et les parcourir lorsqu'ila besoin de la valeur d'une variable. Cette gestion des environnements est une source d'inefficacité mais elle est intéressante car elle définit une fonction que le code compilé devra simuler.

Dans une implantation compilée, le contrôle est défini par la suite des instructions à exécuter. La syntaxe du programme source a disparu et le code compilé incorpore le contrôle qui lui était associé. La suite des instructions n'est toutefois pas linéaire. Elle contient des instructions de branchement utilisées en particulier pour traduire les appels de fonctions contenues dans le programme source. Après l'exécution d'un appel de fonction, on doit revenir au code appelant et ceci se fait en mémorisant le point de retour dans une pile d'exécution. La pile est l'outil essentiel de gestion du contrôle.

La gestion des données suppose l'utilisation de mémoire pour stocker ces données. La façon dont on doit allouer cette mémoire dépend de la durée de vie que l'on doit accorder aux données correspondantes. Les arguments des fonctions et les valeurs des variables locales ont une durée de vie limitée qui correspond à l'évaluation de la fonction ou du bloc où ils apparaissent. Ils peuvent donc être stockés dans la pile d'exécution. Par exemple, dans le code compilé correspondant à un appel de fonction, on placera des instructions qui empilent non seulement l'adresse de retour mais aussi les valeurs des arguments et ces valeurs disparaitront lorsque l'appel sera terminé. La pile de contrôle est donc aussi utilisée comme pile de données pour ces valeurs temporaires. Le compilateur traduit les utilisations de variables locales ou

des paramètres de fonctions qui se trouvent dans le code source par des instructions qui vont extraire des valeurs de la pile : il doit pour cela précalculer à quelle distance du sommet de pile ces valeurs se trouveront lorsque le code compilé en aura besoin.

Un programme manipule aussi des données dont la durée de vie est indépendante du contrôle. Il s'agit des données correspondant aux variables globales du programme et des données créées dynamiquement par des instructions telles que malloc en C ou new en JAVA. Les premières posent peu de problèmes : le compilateur calcule la taille mémoire qui leur est nécessaire et celle-ci est allouée dans la zone statique du processus qui va exécuter le code compilé. Par contre, la création dynamique de données est un exemple de fonctionnalité plus complexe pour lesquelles le code compilé peut avoir besoin de s'appuyer sur un environnement d'exécution plus élaboré. Ce problème est traité dans la section suivante.

1.5 Les environnements d'exécution

1.5.1 Gestion dynamique de la mémoire

Tous les langages modernes autorisent la création dynamique de structures de données. Ils mettent pour celà à la disposition du programmeur des fonctions comme par exemple `malloc` en C qui permettent de réserver dans une zone mémoire dite dynamique (le tas) de l'espace pour placer des structures de données qui seront référencés par des pointeurs. Cela suppose la mise en place dans tout processus en cours d'exécution de dispositifs permettant d'allouer et aussi de libérer de la mémoire. C'est le service minimum que fournissent C, C++ et PASCAL.
Les langages JAVA, CAML et SCHEME font beaucoup plus. Dans ces langages, il n'est pas nécessaire d'allouer explicitement de la mémoire pour construire dynamiquement de nouvelles valeurs ou de nouveaux objets et par ailleurs, il n'est pas nécessaire de manipuler explicitement des pointeurs pour y avoir accès. Dans ces trois langages, toute valeur est représentée dans un mot mémoire qui contient soit ce qu'on appelle une valeur immédiate c'est-à-dire une valeur qui tient effectivement dans un mot mémoire comme par exemple un nombre, soit un pointeur vers une donnée allouée dans la zone dynamique (le tas). Cette différence entre valeur immédiate et valeur pointée est transparente pour l'utilisateur. Il n'a pas à s'en préoccuper.
En JAVA, une allocation a lieu quand on construit un nouvel objet en utilsant le mot-clé `new`. Ce mot-clé est le mêne que celui de PASCAL mais il ne s'applique pas à un pointeur comme en PASCAL mais à un nom de classe.
En CAML, une allocation a lieu quand on utilise un constructeur de valeur. Il n'y a pas de mot-clé comme `new`. On applique simplement un constructeur à des arguments. Par exemple, on peut écrire `(x,y)` où la virgule est le

constructeur de paires. Cette écriture entraine l'allocation d'un doublet (une paire de mots) dans le tas et ces deux mots reçoivent respectivement la valeur de `x` et la valeur de `y`.
En SCHEME, quand on utilise le mécanisme de quotation pour construire une structure de données à partir d'une expression, l'allocation se fait aussi de façon transparente.

Outre l'absence d'utilisation de pointeur explicite, la principale différence entre JAVA, CAML et SCHEME d'une part et C et PASCAL d'autre part, c'est que les trois premiers récupèrent aussi automatiquement la mémoire qui n'est plus utilisée grâce à un **garbage collector**. Le système de gestion de la mémoire est capable de détecter les structures qui ne sont plus référencées et de récupérer la place mémoire qu'elles occupaient pour allouer de nouvelles structures. Le programmeur n'a pas à effectuer cette gestion lui-même, ce qui allège la programmation et évite de nombreuses erreurs.
Dans ces langages, le système d'exécution (runtime system) est donc nettement plus complexe qu'en C ou PASCAL puisqu'il comporte ce mécanisme complexe d'allocation et de récupération de la mémoire.

1.5.2 Typage dynamique

Le langage SCHEME est un exemple pur de langage non typé. Non seulement il n'y a pas en SCHEME de vérification statique de types mais il n'y a pas véritablement de types non plus. Le langage SCHEME a une structure de données unique, essentiellement des arbres binaires. Toutes les données traitées par un programme sont codées dans cette structure de données unique et elles ne sont distinguables les unes des autres que dans l'intention du programmeur et éventuellement les commentaires qui accompagnent le programme.
Toutefois, à l'intérieur de cette structure de données unique, on distingue tout-de-même quelques catégories de valeurs.
– Les symboles (qui jouent entre autres le rôle d'identificateurs)
– les nombres
– les caractères
– les chaînes de caractères
– les booléens
– les vecteurs
– les paires
Ces différentes catégories de valeurs sont distinguables dynamiquement (à l'exécution) grâce à des fonctions de tests `symbol?`, `number?`, `char?`, `string?`, `boolean?`, `vector?` et `pair?`. On peut donc considérer ces catégories comme des types dynamiques, c'est-à-dire des types présents à l'exécution. C'est au programmeur de s'assurer, avant d'appliquer une fonction, que ses arguments sont corrects. Il utilise pour celà les fonctions de test mentionnées plus haut.

En résumé, on peut considérer que SCHEME est le plus polymorphe des langages puisqu'aucune vérification n'est faite. Il peut appliquer n'importe quelle fonction à n'importe quel argument. La contrepartie est que le programmeur ne peut espérer aucune aide du compilateur ou de l'interpréteur. C'est le prix de la liberté absolue.

1.5.3 Interprétation dynamique de messages

On retrouve ce type de situation dans certains langages à objets comme OBJECTIVE C. Ce langage possède pourtant des types statiques comme C++ ou JAVA. Il est possible d'utiliser ces types de façon statique et d'obtenir les même garanties que si on programmait dans ces langages. Mais il est aussi possible de "débrancher" la vérification statique. Pour celà, on dispose d'un type spécial noté `id` qui représente le type "pointeur sur un objet quelconque". Si on introduit une variable `x` par la déclaration

```
id x;
```

le compilateur vérifie simplement que `x` est utilisée de façon cohérente comme un pointeur sur un objet mais par contre, lorsqu'on envoie un message à `x` comme par exemple

```
[x display];
```

le compilateur ne peut pas vérifier que l'objet pointé par `x` connait bien la méthode `display`. En contrepartie, OBJECTIVE C fournit des méthodes permettant d'interroger dynamiquement un objet sur les classes auxquelles il appartient ou sur les méthodes qu'il connait. Les objets déclarés avec le type `id` sont donc traités comme les valeurs de SCHEME .

Le fait de savoir si une telle liberté est nécessaire pour exploiter toute la puissance de la programmation orientée objets est un problème intéressant sur lequel nous reviendrons. Mais on peut tout de suite noter que même les langages à objets qui n'autorisent pas ce type de liberté ont besoin d'une certaine forme de dynamisme. Sopposons par exemple qu'on ait défini une classe `A` puis une sous-classe `B` de `A` et que ces deux classes possèdent une méthode `display` permettant de visualiser les objets de `A` et de `B` sur l'écran. Si on a pris la peine de redéfinir dans `B` la méthode `display` alors que `B` héritait de la méthode `display` de `A`, c'est certainement que les objets de `B` possèdent des attributs supplémentaires qui sont utiles pour la visualisation. Un objet de classe `B` doit donc être visualisé avec la méthode `display` de `B` et non pas avec celle de `A`. Supposons maintenant qu'on gère un tableau d'objets pouvant appartenir à la classe `A` ou à la classe `B`. Ce tableau devra être déclaré statiquement comme un tableau d'objets de classe `A`. Par exemple, en JAVA, on écrira :

```
A[] tab;
```

Maintenant, si on veut visualiser simultanément tous les objets contenus dans le tableau `tab`, on écrira :

```
for (i=0; i<tab.length; i++)
   tab[i].display();
```

et on voudra que la méthode `display` utilisée soit celle de `A` ou celle de `B` suivant la classe de `tab[i]`. Comme il n'y a aucun moyen de connaitre statiquement la classe des éléments du tableau, le choix de la méthode appelée ne peut se faire que dynamiquement. C'est ce que fait JAVA et c'est ce qu'on peut aussi obtenir en C++ en déclarant la méthode `display virtual`. Tous les langages à objets comportent donc une part de dynamisme qui a des implications sur leur environnement d'exécution.
En JAVA par exemple et aussi en OBJECTIVE C, les classes ont une existence dynamique à l'exécution. Elles sont représentées par un objet appelé **objet de classe** (class object) qui a la responsabilité de créer de nouvelles instances et de contenir les méthodes utilisées soit par les instances soit par la classe elle-même. Ces objets sont chaînés entre eux. Un objet de classe pointe vers l'objet de sa super classe, ce qui permet de consulter la hiérarchie des classes dynamiquement et d'implanter la liaison dynamique des méthodes.

Chapitre 2

Analyse syntaxique

L'analyse syntaxique est un domaine qui a fait l'objet de nombreux travaux de recherche. Elle utilise des techniques nombreuses et assez différentes les une des autres. Nous ne pouvons pas les présenter ici de façon complète. Nous nous contenterons de donner les informations permettant de comprendre les analyseurs montants de type LALR(1) [2] et d'utiliser le générateur d'analyseurs syntaxiques YACC dont l'interface avec CAML sera décrite dans le chapitre 3. Nous supposons connues les notions d'automate fini et d'expression régulière (rationnelles) ainsi que les notions de grammaire à règles hors-contexte (context-free) et d'arbre de dérivation. Nous rappelons malgré tout quelques définitions et fixons certaines notations.

2.1 Quelques rappels et notations

2.1.1 grammaires

Pour définir une grammaire, on a besoin de deux alphabets :
– Un alphabet non-terminal N
– Un alphabet terminal T

Un **règle** de grammaire est de la forme

$$S \rightarrow w \qquad \text{où} \qquad S \in N\ ,\ w \in (N \cup T)^*$$

On écrit $(S \rightarrow w) \in G$ lorsque la règle appartient à une grammaire G.

Le symbole $\rightarrow$ sert aussi à noter la **relation de dérivation immédiate** définie par une grammaire G. Pour $u, v \in (N \cup T)^*$, on écrit

$$u \rightarrow v \quad \text{lorsque} \quad u = u_1 S u_2\ ,\ v = u_1 w u_2,\ \text{et } (S \rightarrow w) \in G.$$

On note $\xrightarrow{*}$ la **relation de dérivation** associée à une grammaire (fermeture réflexive et transitive de la relation de dérivation immédiate).

Enfin, le **langage engendré** par une grammaire G à partir de l'un de ses non-terminaux S est

$$L_G(S) \ = \ \{ \ u \in T^* \ / \ S \stackrel{*}{\rightarrow} \ u\}$$

2.1.2 un exemple

$$G \qquad \left\{ \begin{array}{l} S \ \rightarrow \ aSS \\ S \ \rightarrow \ b \end{array} \right.$$

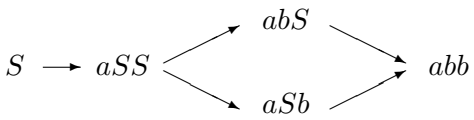
Le langage engendré par cette grammaire est le langage des expressions écrites en notation polonaise préfixe avec un opération préfixe binaire (a) et une constante (b). Il est caractérisé par les propriétés suivantes :

1. Dans un mot de ce langage, le nombre de b est supérieur d'une unité au nombre de a.
2. Dans un facteur gauche d'un mot de ce langage, le nombre de a est supérieur ou égal au nombre de b.

Bien que très simple, ce langage nous sera utile dans la présentation des principes de l'analyse syntaxique.

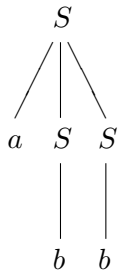
2.1.3 arbres de dérivation et ambiguité

Lorsqu'un mot est engendré par une grammaire, il peut l'être de plusieurs façons. Par exemple, dans notre grammaire exemple, le mot abb peut être obtenu par les deux dérivations :



Ces deux dérivations ne diffèrent que par l'ordre dans lequel on récrit les non-terminaux. La dérivation du haut est une dérivation **gauche** : le non-terminal récrit à chaque étape est le non-terminal le plus à gauche. La dérivatiopn du bas est une dérivation **droite** : le non-terminal récrit à chaque étape est le non-terminal le plus à droite.

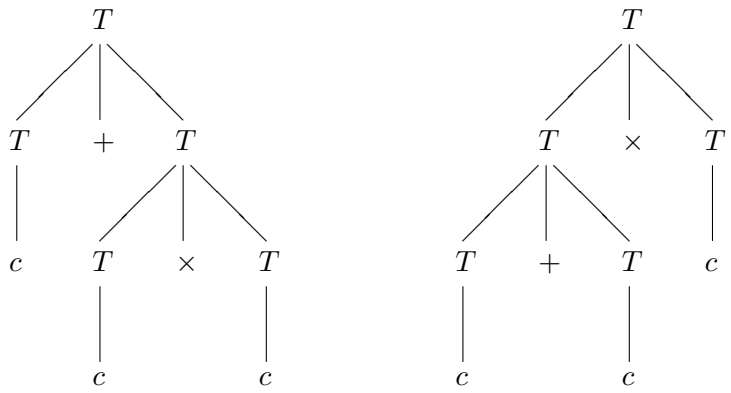
Ces deux dérivations diffèrent mais de façon inessentielle : elles correspondent toutes deux au même **arbre de dérivation** qui est représenté ci-dessous.



Une grammaire qui possède un seul arbre de dérivation pour tout mot qu'elle engendre est dite **non-ambiguë**. Les grammaires des langages de programmation doivent être non-ambiguës pour permettre l'existence d'analyseurs syntaxiques dont la fonction est essentiellement de reconstruire l'arbre de dérivation (unique) correspondant à un programme. Cependant, on peut aussi utiliser des grammaires ambiguës à condition que l'ambiguité puisse être levée par des conventions extérieures à la grammaire. Le cas typique est celui des expressions arithmétiques écrites avec des opérateurs infixes (écrits entre leurs deux arguments). La grammaire

$$\begin{aligned} T &\rightarrow T + T \\ T &\rightarrow T \times T \\ T &\rightarrow c \end{aligned}$$

est ambiguë car l'expression $c + c \times c$ possède les deux arbres de dérivation



Cependant, les priorités respectives des opérateurs + et × permettent de choisir l'arbre de gauche. Dans YACC, l'utilisateur a la possibilité de définir ce type de priorité pour lever des ambiguités.

2.2 Analyse syntaxique montante

2.2.1 automates SHIFT / REDUCE

L'analyse syntaxique montante construit un arbre de dérivation en procédant du bas vers le haut, c'est-à-dire des feuilles vers la racine. On utilise pour cela un automate à pile dont la pile contient des symboles terminaux et non-terminaux. Nous représenterons les configurations de cet automate comme des couple de mots. Le mot de gauche représente la pile dont le sommet est à droite. Le mot de droite est le mot à reconnaitre dont le début est à gauche. Il pourra être commode de marquer la fin du mot d'entrée par un symbole spécial #. Voici un exemple de configuration :

Pile	,	**Entrée**
aSa	,	$bb\#$

Nous commencerons par décrire ces automates comme de simples reconnaisseurs qui testent si un mot appartient au langage engendré par une grammaire donnée. Ensuite, nous ajouterons la production de l'arbre de dérivation.
Les automates d'analyse montante utilisent deux types de mouvements que l'on désigne habituellement sous les noms de SHIFT et REDUCE. Un SHIFT consiste à prendre le premier symbole du mot en entrée et à le placer au sommet de pile. Un REDUCE consiste à reconnaitre au sommet de la pile un mot qui est un membre droit de règle et à le remplacer par le membre gauche de cette règle.

Décrit ainsi, ce fonctionnement est non-déterministe pour deux raisons différentes :

1. Les conflits SHIFT/REDUCE
   Tant que le mot d'entrée possède des symboles, il est toujours possible de faire un SHIFT . Même lorsqu'un REDUCE. est possible, c'est-à-dire lorsque le haut de pile contient un membre droit de règle, il peut être utile dans certains cas de faire plutôt un SHIFT.
2. Les conflits REDUCE/REDUCE
   Le haut de pile peut contenir deux membres droits de règles, l'un étant facteur droit de l'autre. Dans ce cas, on ne sait pas quelle réduction privilégier.

Les techniques de construction d'analyseurs montants doivent permettre de déterminiser ces automates en faisant un choix pour chaque conflit possible.

2.2.2 exemple

Reprenons l'exemple :

$$G \qquad \begin{cases} S \rightarrow aSS \\ S \rightarrow b \end{cases}$$

L'automate d'analyse montante correspondant ne comporte pas de conflit REDUCE/REDUCE car aucun membre droit de règle n'est facteur droit d'un autre. Quant aux conflits SHIFT/REDUCE, on les règle en privilégiant systématiquement les REDUCE par rapport aux SHIFT. Un mot est reconnu si on arrive à une configuration où le mot d'entrée est complêtement lu (il n'y a plus en entrée que le symbole #) et la pile contient comme unique symbole le non-terminal S. Notons que le symbole spécial # ne fait jamais l'objet d'un SHIFT.

Pile	,	**Entrée**	
	,	$ababb\#$	
a	,	$babb\#$	SHIFT
ab	,	$abb\#$	SHIFT
aS	,	$abb\#$	REDUCE
aSa	,	$bb\#$	SHIFT
$aSab$	,	$b\#$	SHIFT
$aSaS$	,	$b\#$	REDUCE
$aSaSb$	,	$\#$	SHIFT
$aSaSS$	,	$\#$	REDUCE
aSS	,	$\#$	REDUCE
S	,	$\#$	REDUCE

2.2.3 dérivation droite inverse

Les automates à pile déterministes correspondant à des analyseurs montants SHIFT/REDUCE sont souvent désignés, en particulier dans la littérature anglo-saxonne par le terme d'automates LR. Dans cette désignation, le L (Left) fait référence au fait que les mots d'entréee sont lus à partir de la gauche et le R (Right) fait référence au fait qu'ils reconstituent une dérivation droite. Cette dérivation droite est reconstituée à l'envers, c'est-à-dire que la première action de type REDUCE correspond à la dernière règle appliquée dans la dérivation.
En effet, si nous effaçons la séparation entre la pile et l'entrée et effaçons le marqueur de fin de mot, le calcul précédent peut s'écrire simplement comme une dérivation inverse, c'est-à-dire comme un calcul qui part d'un mot engendré par une grammaire et qui applique les règles de la grammaire à l'envers pour aboutir au non-terminal initial.

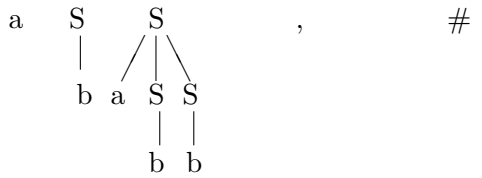
$$
\begin{array}{c}
ababb \\
aSabb \\
aSaSb \\
aSaSS \\
aSS \\
S
\end{array}
$$

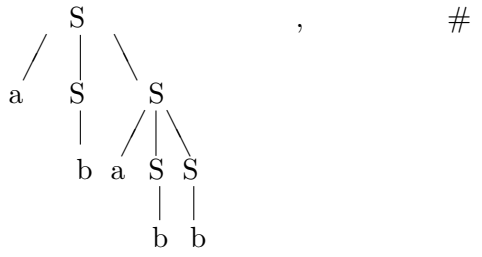
Lue du bas vers le haut, cette suite d'étapes est une dérivation droite.

2.2.4 construction de l'arbre

Revenons au calcul de la section 2.2.2. Chaque symbole non-terminal qui se trouve sur la pile a été obtenu par un mouvement de type REDUCE et on peut associer à ces symboles les morceaux d'arbres de dérivation qui correspondent à leur apparition. A la fin du calcul de l'automate de reconnaissance, le non-terminal restant sur la pile sera associé à l'arbre de dérivation complet. Nous reprenons le calcul précédent en ajoutant la construction de l'arbre.

	,	ababb#
a	,	babb#
ab	,	abb#
aS | b	,	abb#
aSa | b	,	bb#
aSab | b	,	b#
aSaS | | b b	,	b#
aSaSb | | b b	,	#
aSaSS | | | b bb	,	#





2.3 Les tables d'analyse montantes

En général, quand on part d'une grammaire non-ambiguë quelconque, il est impossible d'arbitrer les conflits SHIFT/REDUCE ou REDUCE/REDUCE de façon à obtenir un automate d'analyse montante déterministe. Les techniques de construction de tables que nous allons présenter ici et qui sont utilisées par exemple par YACC, le permettent dans certains cas. Cependant, il n'existe aucune caractérisation agréable des grammaires qui conduisent à des analyseurs montants déterministes. En général, celui qui veut obtenir un analyseur syntaxique montant pour un langage donné procède ainsi :

1. En s'appuyant sur son expérience et/ou sur la connaissance de grammaires définissant d'autres langages, il écrit une grammaire pour son langage et la code dans le format accepté par YACC.
2. Il appelle ensuite YACC qui construit des tables d'analyse et met éventuellement en évidence des conflits SHIFT/REDUCE ou REDUCE/REDUCE
3. Le programmeur corrige alors sa grammaire et la resoumet à YACC . Ce processus est itéré jusqu'à l'obtention d'une grammaire satisfaisante et donc d'un analyseur déterministe.

Bien que cette procédure soit loin d'être complêtement satisfaisante, elle est souvent utilisée car l'expérience montre qu'elle permet de construire des analyseurs déterministes pour la plupart des langages de programmation et que les analyseurs construits sont efficaces.

2.3.1 FIRST et FOLLOW

La déterminisation des automates montants SHIFT/REDUCE repose en partie sur la connaissance des symboles terminaux qui peuvent suivre un non-terminal S dans un mot intervenant au cours d'une dérivation d'une grammaire donnée. L'ensemble de ces symboles est noté FOLLOW(S). Si S_0 est le non-terminal initial de la grammaire considérée,

$$FOLLOW(S) = \{a \in T \mid \exists u, v \in (N \cup T)^* \ S_0 \xrightarrow{*} uSav\}$$

Pour calculer FOLLOW(S), il faut savoir aussi calculer l'ensemble des premiers symboles possibles des mots engendrés à partir d'un non-terminal ou plus généralement d'une suite de non-terminaux et de terminaux. On note cet ensemble FIRST(S) OU FIRST(w). Dans le cas où le mot vide ϵ peut être produit à partir de S ou w, on l'inclut aussi dans FIRST.

$$FIRST(w) = \{a \in T \mid \exists u \in (N \cup T)^* \ w \xrightarrow{*} au\} \ \ (\cup \{\epsilon\} \ si \ w \xrightarrow{*} \epsilon \ )$$

Pour calculer $FIRST$, on applique les règles suivantes :

1. Si X est un symbole terminal, FIRST(X)=$\{X\}$.
2. Si $X \to \epsilon$ est une régle, $\epsilon \in$ FIRST(X).
3. Si $X \to Y_1 \dots Y_n$ est une régle, pour tout symbole terminal a, si $\exists i \ \ a \in$ FIRST(Y_i) et $\forall j < i \ \ \epsilon \in$ FIRST(Y_j), alors $a \in$ FIRST(X). Par ailleurs, si $\forall i \ \ \epsilon \in$ FIRST(Y_i), alors $\epsilon \in$ FIRST(X).

Enfin, $a \in$ FIRST($X_1 \dots X_n$) si $a \in$ FIRST(X_1) ou $\epsilon \in$ FIRST(X_1) et $a \in$ FIRST($X_2 \dots X_n$) .

Pour calculer FOLLOW, on applique les règles suivantes :

1. Par convention, le marqueur de fin de mots # appartient à FOLLOW(S_0) si S_0 est le non-terminal principal considéré.
2. Si $T \to \alpha U \beta$ est une règle, alors FOLLOW(U) contient tous les symboles terminaux appartenant à FIRST(β).
3. S'il existe une règle $T \to \alpha U$ ou bien une règle $T \to \alpha U \beta$ avec $\epsilon \in$ FIRST(β), alors FOLLOW(T) $\subseteq$ FOLLOW(U) .

Notons que ce calcul de FOLLOW suppose que la grammaire est réduite c'est-à-dire que tout non-terminal peut être atteint et engendre un langage non-vide.

2.3.2 Utilisation de FIRST et FOLLOW

Considérons la grammaire :

$$G \quad \begin{cases} S \rightarrow aT \\ S \rightarrow bU \\ T \rightarrow bc \\ U \rightarrow b \end{cases}$$

La reconnaissance du mot abc commence par :

Pile	,	**Entrée**	
	,	$abc\#$	
a	,	$bc\#$	SHIFT
ab	,	$c\#$	SHIFT

A l'étape suivante, il est possible de faire un REDUCE de la règle $U \rightarrow b$ mais ce choix conduit ensuite aux étapes

aU	,	$c\#$	REDUCE
aUc	,	$\#$	SHIFT

et la reconnaissance échoue. Il est possible de voir que l'étape REDUCE est un mauvais choix en examinant FOLLOW(U). Pour que cette étape puisse conduire à la reconnaissance du mot de départ, il faudrait que FOLLOW(U) contienne c. Or FOLLOW(U) = $\{\#\}$.

Le bon calcul est :

abc	,	$\#$	SHIFT
aT	,	$\#$	REDUCE
S	,	$\#$	REDUCE

et le choix des deux étapes REDUCE peut être justifié par le fait que FOLLOW(T) = FOLLOW(S) = $\{\#\}$.

2.3.3 Les automates et tables SLR

La construction d'analyseurs LR utilise deux outils pour régler les conflits et aboutir à un automate à pile déterministe :

1. Un automate fini est utilisé pour analyser à tout instant le contenu de la pile. Cet automate permet de savoir si un mouvement REDUCE est possible sans avoir pour cela à explorer la pile.
2. L'action à exécuter à un moment donné, SHIFT ou REDUCE, dépend de l'état de cet automate fini et du premier symbole en entréee. Elle est donnée par une table à double entrée appelée table d'analyse LR.

Pour être complet, il faut mentionner que, dans leur version la plus générale, les tables d'analyse LR prennent en compte les k premiers symboles en entrée et pas seulement le premier. On parle alors d'analyseurs LR(k). Toutefois, dans les outils LR couramment disponibles, et en particulier dans YACC,

on a $k = 1$. Nous utiliserons donc simplement LR pour dire LR(1) et nous donnerons la construction des automates et des tables uniquement pour ce cas.

L'automate fini utilisé pour analyser le contenu de la pile est conçu pour indiquer à tout instant si une ou plusieurs règles peuvent être contractée par un mouvement de type REDUCE et lesquelles. Les états de l'automate fini sont intercalés entre les symboles de pile pour former une pile de la forme

$$p_0 Y_1 q_1 \dots q_{n-1} Y_n q_n$$

L'état q_i est l'état atteint par l'automate fini après la lecture du mot $Y_1 \dots Y_i$. L'état q_n est l'état atteint par l'automate fini après la lecture de toute la pile. C'est en fonction de l'état q_n et de la première lettre en entrée que la décision de faire un SHIFT ou un REDUCE sera prise et que le choix éventuel entre plusieurs REDUCE sera tranché.

Les états de ces automates finis sont constitués d'ensembles d'**items**. Un item est une règle de la grammaire où une position est marquée (par un point) dans le membre droit. Par exemple, à une règle comme $S \to aSS$ correspondent quatre items :

$$[S \to .aSS] \ , \ [S \to a.SS] \ , \ [S \to aS.S] \ , \ [S \to aSS.]$$

La règle $S \to aSS$ sera susceptible d'être réduite dans un mouvement de type REDUCE si l'état q_n qui se trouve au sommet de pile contient l'item $[S \to aSS.]$.

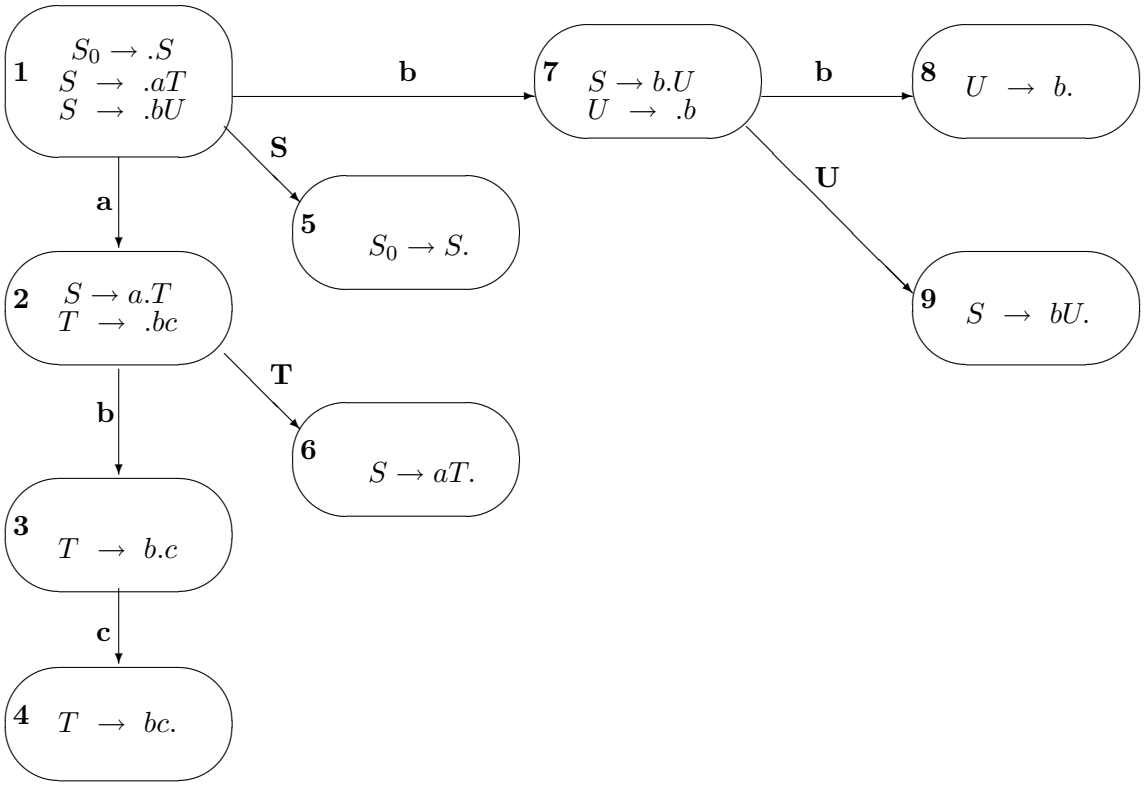
L'état initial de l'automate contient tous les items de la forme $[S_0 \to .w]$ où S_0 est le non-terminal initial de la grammaire et $S_0 \to w$ est une règle de la grammaire. C'est l'état dans lequel on se trouve quand on n'a encore lu aucun caractère en entrée : on sait que le premier non-terminal utilisé pour dériver le mot d'entrée est nécéssairement S_0 mais on ne sait pas quelle régle ayant S_0 pour membre gauche doit être utilisée.

Chaque ensemble d'items, et en particulier celui constituant l'état initial doit être fermé par la construction suivante : si un état contient un item $[S \to u.Tv]$, il doit aussi contenir tous les items de la forme $[T \to .w]$ où $T \to w$ est une règle de la grammaire. L'idée est que si les caractères qu'on a lu jusque-là rendent possible le fait qu'on se trouve juste avant le caractère T dans le membre doit de la règle $S \to uTv$, alors on peut être aussi tout au début du membre droit de toute règle $T \to w$.
Il est commode d'ajouter un nouveau non-terminal initial S_0 à la grammaire et la règle $S_0 \to S$ où S est l'ancien non-terminal initial. De cette façon, toute reconnaissance se termine par la contraction de cette nouvelle règle.

Ensuite, il faut décrire les transitions de l'automate fini. Pour tout symbole de pile Y (qui peut être un terminal ou un non-terminal de la grammaire),

si un état contient un item de la forme $[T \to u.Yv]$, alors l'état auquel on arrive par la transition correspondant à Y devra contenir l'item $[T \to uY.v]$. Cet état devra aussi être fermé au sens défini précédemment.
Par exemple, on trouvera ci-dessous l'automate correspondant à la grammaire

$$G \quad \begin{cases} S \to aT \\ S \to bU \\ T \to bc \\ U \to b \end{cases}$$



Cet automate fournit beaucoup d'informations concernant les mouvements SHIFT et REDUCE. Un SHIFT d'une lettre terminale a est envisageable s'il existe une transition étiquetée par a à partir de cet état. Un REDUCE est envisageable si l'état du sommet de pile contient un item de la forme $[T \to w.]$. Cette dernière information doit toutefois être complétée par celle donnée par FOLLOW(T). On se base donc sur l'automate pour construire une table prenant en lignes les états de l'automate et en colonnes les symboles d'entrée et en indiquant dans chaque case si un SHIFT ou un REDUCE est possible.

– Quand un SHIFT est possible, on l'indique par la lettre s. Comme ce mouvement conduit à ajouter un symbole sur la pile, il faut aussi indiquer le nouvel état correspondant au sommet de pile.
– Quand un REDUCE est possible, il faut indiquer à quelle règle il correspond. On numérote les règles de la grammaire et on indique juste r (pour reduce) et le numéro de la règle.

Enfin, quand on fait un REDUCE, on enlève des symboles en haut de la pile et on les remplace par un non terminal T. Là aussi, il faut calculer le nouvel état de sommet de pile à partir de l'état que se trouve juste en-dessous et du non-terminal T. Cette information est fournie par la transition de l'automate correspondant à l'ancien état et à T.

Numerotation des regles
$$\begin{cases} 1: S_0 \rightarrow S \\ 2: S \rightarrow aT \\ 3: S \rightarrow bU \\ 4: T \rightarrow bc \\ 5: U \rightarrow b \end{cases}$$

	FIRST	FOLLOW
S_0	a b	#
S	a b	#
T	b	#
U	b	#

Les informations fournies par l'automate et le calcul de FOLLOW permettent de construire finalement les tables appelées ACTION et GOTO. La table ACTION indique pour chaque état de l'automate et chaque symbole d'entrée si on doit effectuer un SHIFT et alors, quel est le nouvel état du sommet de pile ou bien un REDUCE et alors le numéro de la règle à réduire. La table GOTO permet de recalculer l'état du sommet de pile après un REDUCE .

Si toutes les cases de la table ACTION contiennent au plus une action, alors on obtient un automate déterministe pour la reconnaissance du langage et on dit que la grammaire de départ est SLR. Malheureusement, il n'existe pas d'autre méthode pour tester si une grammaire est SLR que la construction de la table. On doit donc disposer d'une implantation de la méthode que nous venons de voir. C'est ce que fournit notamment YACC. En fait YACC. utilise une notion de table un peu plus générale que les tables SLR (Les tables LALR) et fournit aussi la possibilité de déclarer des propriétés supplémentaires comme les priorités respectives des opérateurs. Mais les tables construites restent proches des tables SLR.

Ces techniques de construction de tables d'analyse syntaxique montante ont une propriété propriété supplémentaire qui est assez agréable. Si la table obtenue n'est pas déterministe, il est toujours possible de faire un choix arbitraire pour les conflits qu'elle contient. Dans le cas d'un conflit REDUCE/REDUCE, on peut choisir la règle à réduire en fonction de l'ordre des règles de la grammaire. Dans le cas d'un conflit SHIFT/REDUCE on peut privilégier le SHIFT. Ce sont les choix que fait YACC. Si on effectue de tels choix, on obtient un analyseur qui ne reconnait peut-être pas tout le langage engendré par la grammaire mais qui reconnait en tout cas un sous-ensemble de ce langage.

	ACTION				GOTO		
	a	b	c	#	S	T	U
1	s2	s7			5		
2		s3				6	
3			s4				
4				r4			
5				accept			
6				r2			
7		s8					9
8				r5			
9				r3			

Le calcul de reconnaissance du mot abc se déroule comme suit :

Pile	,	**Entrée**	
1	,	$abc\#$	
$1a2$	,	$bc\#$	S2
$1a2b3$	,	$c\#$	S3
$1a2b3c4$	,	$\#$	S4
$1a2T6$	,	$\#$	R4 GOTO6
$1S5$	,	$\#$	R2 GOTO5
		ACCEPT	

2.4 Traitement des expressions arithmétiques

Dans cette section, nous traitons le cas des expressions arithmétiques utilisant des opérateurs infixes binaires. Pour voir l'ensemble des problèmes posés par ces expressions, il faut au moins considérer deux opérateurs de priorités différentes et nous prendrons donc les opérateurs d'addition et de multiplication. La première grammaire (ambiguë) de notre langage sera :

$$\begin{cases} E \rightarrow E + E \\ E \rightarrow E \times E \\ E \rightarrow c \\ E \rightarrow (E) \end{cases}$$

2.4.1 Traitement de l'ambiguité

Le fait que l'opérateur de multiplication a priorité sur celui de l'addition peut être pris en compte comme information supplémentaire, donnée hors de la grammaire. YACC, par exemple, donne cette possibilité. Cependant, on peut aussi prendre en compte cette information à l'intérieur de la grammaire, ce qui amène à considérer plusieurs niveaux d'expressions.

- Les **facteurs** : ce sont soit la constante c soit une expressions entourée de parenthèses.
- Les **termes** : ce sont des produits de facteurs ou des facteurs.
- Les **expressions** : ce sont des somme de termes ou des termes

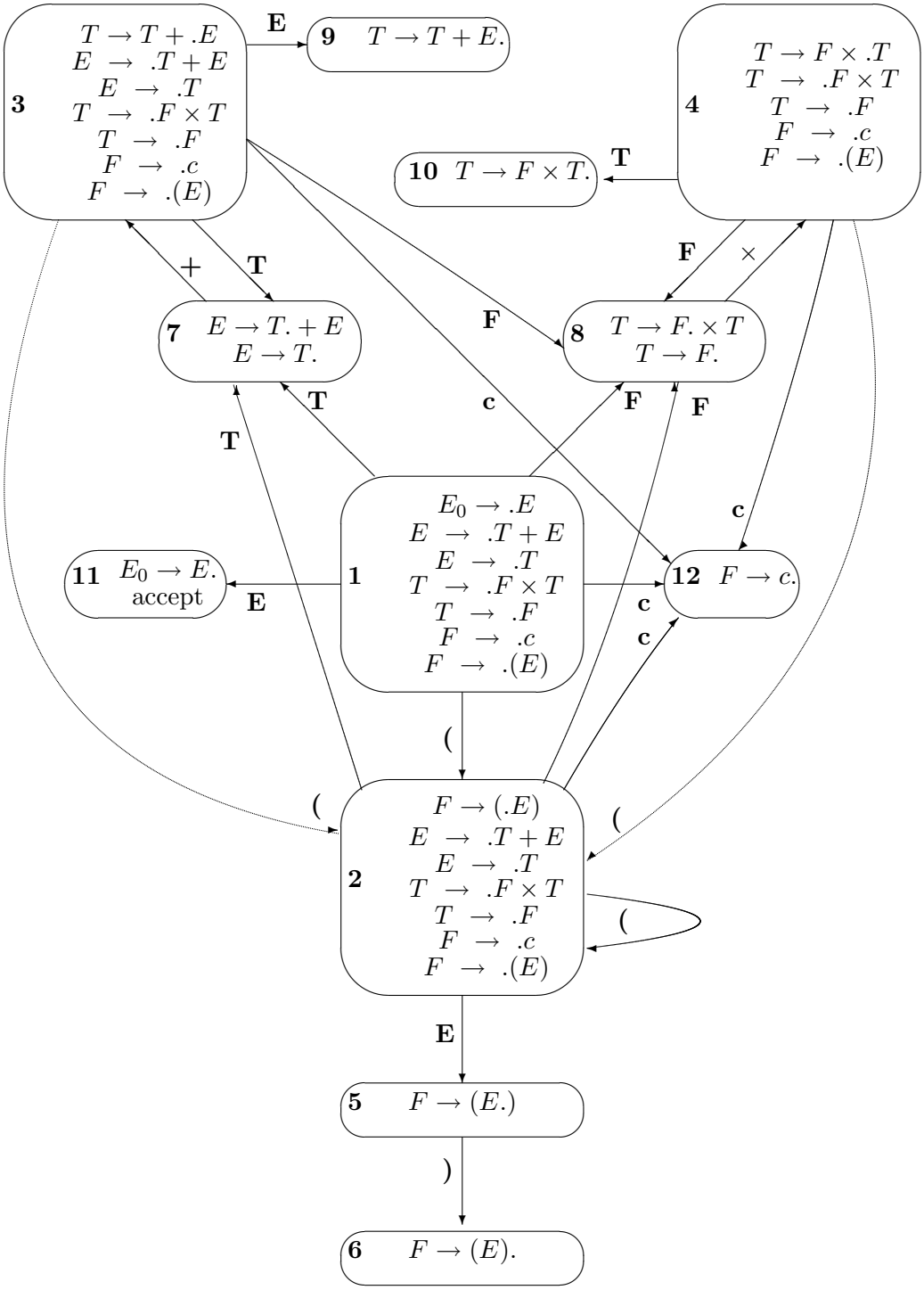
Cette hiérarchisation conduit à la grammaire suivante qui est, cette fois, non-ambiguë.

$$\begin{cases} E \rightarrow T + E \\ E \rightarrow T \\ T \rightarrow F \times T \\ T \rightarrow F \\ F \rightarrow c \\ F \rightarrow (E) \end{cases}$$

Pour construire l'automate et les tables SLR asssociés à cette grammaire, on lui ajoute un non-terminal initial et une règle initiale et on numérote les règles

$$\begin{cases} 1: E_0 \rightarrow E \\ 2: E \rightarrow T + E \\ 3: E \rightarrow T \\ 4: T \rightarrow F \times T \\ 5: T \rightarrow F \\ 6: F \rightarrow c \\ 7: F \rightarrow (E) \end{cases}$$

2.4.2 Construction de l'automate SLR



2.4.3 Construction de la table SLR

	FIRST	FOLLOW
E_0	c (	#
E	c (	) #
T	c (	+) #
F	c (	× +) #

	ACTION						GOTO		
	+	×	(	)	c	#	E	T	F
1			s2		s12		11	7	8
2			s2		s12		5	7	
3			s2		s12			7	8
4		s8	s2		s12			10	8
5				s6					
6	r7	r7		r7		r7			
7	s3			r3		r3			
8	r5	s4		r5		r5			
9	r2			r2		r2			
10	r4			r4		r4			
11						accept			
12	r6	r6		r6		r6			

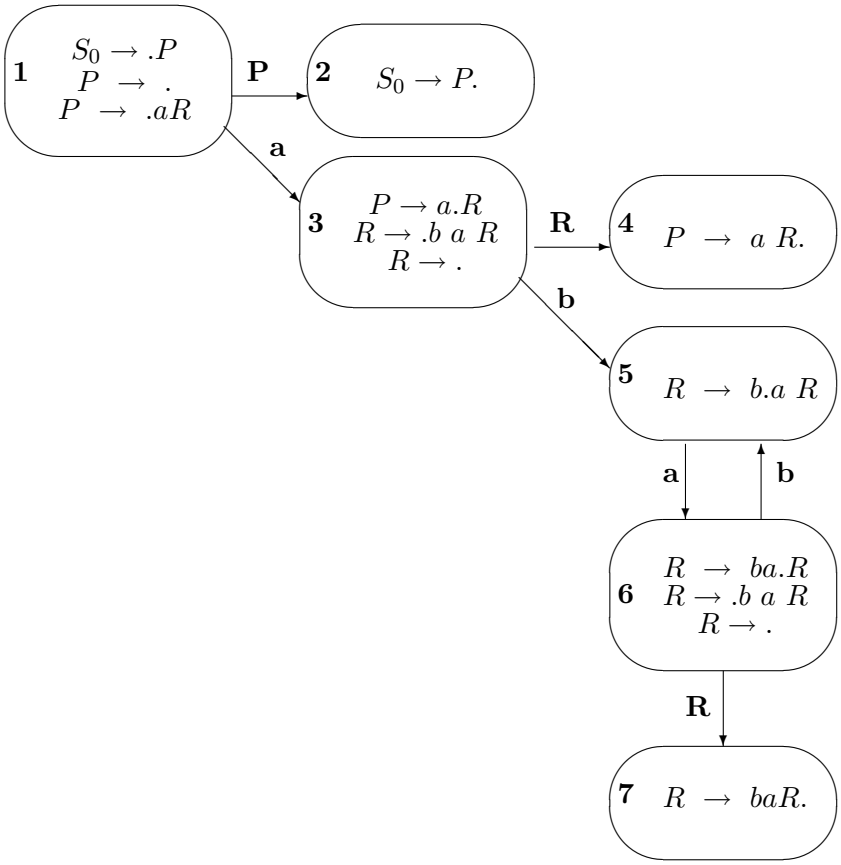
On constate que chaque case de la table des actions contient au plus une action. La grammaire était donc SLR et nous obtenons un analyseur SHIFT/REDUCE déterministe.

2.5 Traitement des listes

Les langages de programmation utilisent souvent des listes dans leur syntaxe. Quand il y a plusieurs objets dans une liste, ils sont en général séparés par un lexèmes appelés séparateurs. Par exemple, dans la liste des arguments d'une fonction, c'est la virgule qui est utilisé comme séparateur.
La syntaxe des listes se présente sous trois forme possibles. Une liste peut être vide, contenir un seul élément ou contenir au moins deux éléments. Dans ce dernier cas seulement, le séparateur doit apparaître. En notant un argument par le symbole a et le séparateur par le symbole b, ceci conduit à la grammaire suivante :

$$\begin{cases} 1.\ S_0 \rightarrow P \\ 2.\ P \rightarrow \epsilon \\ 3.\ P \rightarrow a\ R \\ 4.\ R \rightarrow b\ a\ R \\ 5.\ R \rightarrow \epsilon \end{cases}$$

Cette grammaire ne pose pas de problème pour la construction SLR mais elle est malgré tout intéressante parce qu'elle utilise le mot vide, ce que nous n'avions pas fait jusqu'à présent. Voici l'automate qu'on obtient :



On a FOLLOW(S_0) = FOLLOW(P) = FOLLOW(R) = $\{\#\}$, ce qui conduit à la table SLR suivante :

	ACTION			GOTO	
	a	b	#	P	R
1	s3		r2	2	
2			accept		
3		s5	r5		4
4			r3		
5	s6				
6		s5	r5		7
7			r4		

Voici un exemple d'analyse :

Pile	,	**Entrée**	
1	,	$aba\#$	
$1a3$	,	$ba\#$	s3
$1a3b5$	,	$a\#$	s5
$1a3b5a6$	,	$\#$	s6
$1a3b5a6R7$	,	$\#$	r5 goto7
$1a3R4$	,	$\#$	r4 goto4
$1P2$	,	$\#$	r3 goto2
		ACCEPT	

Dans l'analyse des arguments d'une fonction, dans un langage de programmation, le rôle de marqueur de fin, joué ici par le symbole #, serait tenu par la parenthèse fermante. Le symbole a serait remplacé par le non-terminal de la grammaire correspondant aux expressions.

2.6 Compléments

2.6.1 Propriétés des automates SLR

Quelle est exactement l'information apportée par l'automate SLR associé à une grammaire ? Quelle est la signification des états de cet automate ? Pour le comprendre, il faut d'abord comprendre la signification exacte des items utilisés pour construire les états de cet automate.
Soit une grammaire G et un non-terminal initial S_0. Partant de cette grammaire, on construit l'automate fini SLR associé puis les tables SLR. On obtient un automate à pile SHIFT/REDUCE qui peut être déterministe ou non-déterministe.
La construction SLR garantit la propriété suivante. Supposons qu'en partant d'une configuration initiale $(., u_1u_2\#)$ on arrive à une configuration intermédiaire $(\alpha, u_2\#)$. Si l'état de l'automate fini SLR associé est q pour la pile α et si cet état contient un item de la forme $[T \to w_1.w_2]$, cela donne 3 informations :

1. $\alpha \xrightarrow{*} u_1$
2. La pile est de la forme $\alpha = \alpha' w_1$
3. Il existe un mot terminal u'' et une dérivation droite $S_0 \xrightarrow{*} \alpha u''$

Ceci implique que le mot $u_1 u''$ est engendré par la grammaire et donc que la partie u_1 du mot de départ, celle qui a été lue, ne contient en elle-même aucune erreur. La technique SLR détecte les erreurs dès que possible.

Par ailleurs, dans le cas où une configuration initiale $(., u\#)$ conduit à la configuration $(S_0, \#)$, on a $S_0 \xrightarrow{*} u$ et le mot u est bien un mot engendré par la grammaire.

Enfin, dans le cas particulier où l'état q contient un item de la forme $[T \rightarrow w.]$, la pile est de la forme $\alpha = \alpha' w$. Une réduction de la règle $[T \rightarrow w]$ est alors autorisée.

Réciproquement, si un mot u est engendré par une grammaire G et un non-terminal initial S_0, il existe un calcul de l'automate à pile SHIFT/REDUCE associé qui conduit de la configuration initiale $(., u)$ à la configuration finale $(S_0, \#)$.

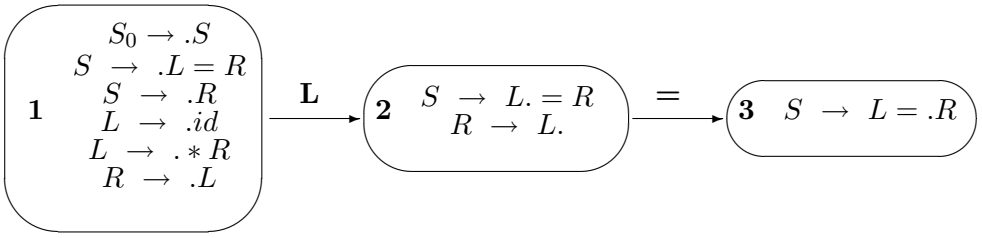
2.6.2 Limites des tables SLR

Pour construire une table d'analyse SLR, on tient compte de l'automate SLR et de FOLLOW. Plus précisément, étant donné un état q de l'automate SLR et un symbole d'entrée a, on autorise dans cette configuration la réduction d'une règle $S \rightarrow w$ si et seulement si q contient l'item $[S \rightarrow w.]$ et a appartient à FOLLOW(S).

En fait, la seule chose dont on est sûr, c'est que si a n'appartient pas à FOLLOW(S), alors il ne faut pas autoriser la réduction de la règle. En revanche, il est tout-à-fait possible que a appartienne à FOLLOW(S) mais que la réduction soit quand même impossible. Voici un exemple (repris de [2]) correspondant partiellement aux affectations du langage C :

$$\begin{cases} S \rightarrow L = R \\ S \rightarrow R \\ L \rightarrow id \\ L \rightarrow *R \\ R \rightarrow L \end{cases}$$

L'automate SLR correspondant à cette grammaire (augmentée par un nouveau non-terminal initial S_0 et une nouvelle règle $S_0 \rightarrow S$) contient en particulier les états et transitions suivants :



Comme le symbole = appartient à FOLLOW(R), l'état **2** permet à la fois un REDUCE et un SHIFT si le symbole en entrée est =. Toutefois, en examinant mieux les raisons qui font que le symbole = appartient à FOLLOW(R), on s'aperçoit que pour trouver le symbole = à gauche d'un R, il faut avoir appliqué la règle $L \rightarrow *R$ et que, dans ce cas, le SHIFT du sumbole = n'est en fait pas possible avant d'avoir réduit cette règle.
On peut améliorer les tables en recueillant un peu plus d'information dans l'automate servant à les construire.

2.6.3 Automates LR et LALR

L'idée est de ne plus se fier à la fonction FOLLOW qui donne pour chaque non-terminal les symboles terminaux qui peuvent le suivre mais d'associer cette information séparément à chaque item. Les nouveau items, que nous appellerons items LR[1], seront des paires formées d'un item au sens précédent et d'un symbole terminal c'est-à-dire quelque chose de la forme $[S \rightarrow u_1.u_2\ ,\ a]$. L'information supplémentaire apportée par le symbole terminal a n'est véritablement utilisée que lorsque le point se trouve à la fin du membre droit c'est-à-dire lorsque l'item est de la forme $[S \rightarrow u.\ ,\ a]$. Dans ce cas, on autorisera une réduction de la règle $S \rightarrow u$ lorsque le symbole d'entrée est a.
Chaque ensemble d'items doit être fermé par une construction de fermeture modifiée comme suit : si un état contient un item $[S \rightarrow u.Tv\ ,\ a]$, il doit aussi contenir tous les items de la forme $[T \rightarrow .w\ ,\ b]$ où $T \rightarrow w$ est une règle de la grammaire et b appartient à FIRST(va).

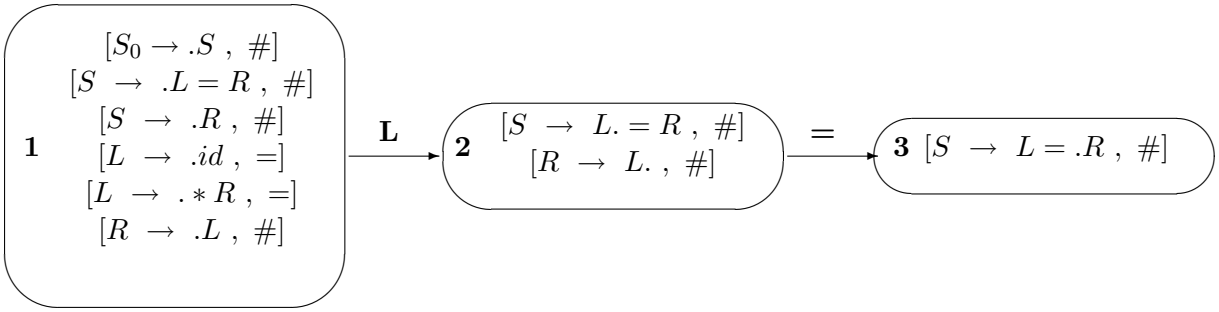
Comme précédemment, pour tout symbole de pile Y (qui peut être un terminal ou un non-terminal de la grammaire), si un état contient un item de la forme $[T \rightarrow u.Yv\ ,\ a]$, alors l'état auquel on arrive par la transition correspondant à Y devra contenir l'item $[T \rightarrow uY.v\ ,\ a]$. Cet état devra aussi être fermé au sens défini précédemment.

Reprenons la grammaire

[1] Dans la littérature, ces items sont plutot désignés sous le nom d'items LR(1) et les précédents sous le nom d'tems LR(0)

$$\begin{cases} S \rightarrow L = R \\ S \rightarrow R \\ L \rightarrow id \\ L \rightarrow *R \\ R \rightarrow L \end{cases}$$

L'ajout d'un symbole terminal dans les items permet d'éliminer le problème de conflit SHIFT/REDUCE car le SHIFT se fait sur le symbole = alors que le reduce correspond au symbole #.



L'automate des items LR permet de construire des tables LR qui présentent moins de conflits que les tables SLR mais il y a un prix à payer. L'ajout de terminaux dans les items a l'inconvénient d'augmenter la taille de l'automate et cette augmentation peut être importante dans le cas de langages de programmation car ceux-ci comportent de nombreux symboles.
En pratique, on utilise plutôt des automates dits LALR dont les états sont obtenus en fusionnant les états des automates LR qui sont les mêmes quand on regarde uniquement la partie SLR des items. On obtient des automates ayant le même nombre d'états que les automates SLR correspondants mais la structure de ces états est plus complexe. Ce sont ces automates LALR qui sont construits par YACC. Dans la plupart des cas, l'automate LALR correspondant à une grammaire est le même que l'automate SLR mais il existe tout de même quelques exemples de grammaires utiles en pratique qui sont LALR mais pas SLR.

2.6.4 Utilisation de grammaires ambiguës

Comme il a déjà été dit plus haut, un des avantages des techniques LR est que, même quand elles échouent à construire directement un analyseur déterministe, elles peuvent être complétées par d'autres moyens de déterminisation. Par exemple YACC, en cas de conflit SHIFT/REDUCE, signale le conflit mais construit quand même un analyseur en arbitrant en

faveur du SHIFT. De même, en cas de conflit REDUCE/REDUCE, il signale le conflit mais construit quand même un analyseur en arbitrant en faveur de la règle qui apparaît en premier dans la liste des règles de la grammaire. L'utilisateur peut donc faire des essais avec un analyseur incomplet mais correct en ce sens qu'il ne reconnaît que des phrases engendrées par la grammaire de départ..
De plus, il est tout à fait possible d'utiliser en complément d'autres techniques. Par exemple, il est facile de prendre en compte dans un analyseur SHIFT/REDUCE la priorité et l'associativité à gauche ou à droite des opérateurs binaires. Si la grammaire fournie donne des rôles semblables à deux opérateurs qui ne sont pas de même priorités comme $+$ et $\times$, par exemple en contenant deux règles $E \to E + E$ et $E \to E \times E$, il y aura un conflit SHIFT/REDUCE dans les deux configurations :

$$wE + E \quad , \quad \times u$$

et

$$wE \times E \quad , \quad + u$$

Dans le premier cas, c'est-à-dire quand l'opérateur en entrée a priorité sur celui qui se trouve en haut de la pile, il faut arbitrer en faveur du SHIFT alors que dans l'autre cas, il faut arbitrer en faveur du REDUCE. Un générateur d'analyseurs comme YACC peut parfaitement prendre la bonne décision si on lui indique les priorités respectives des opérateurs.
De même, dans une configuration où l'opérateur qui se trouve en haut de la pile a même priorité que celui qui se trouve en entrée, une information d'associativité à gauche ou à droite permet de prendre la bonne décision. Si on est dans la configuration

$$wE \oplus E \quad , \quad \oplus u$$

on doit choisir REDUCE si l'opérateur $\oplus$ associe à gauche et SHIFT s'il associe à droite.
Nous verrons au chapitre suivant comment donner ces informations de priorité et d'associativité à YACC.

Chapitre 3

Utilisation de CAML comme métalangage

Comme notre intention dans ce cours est d'étudier de façon précise différents langages de programmation, il sera utile de pouvoir représenter la syntaxe de ces différents langages sous une forme qui peut être directement manipulée, c'est-à-dire dans un langage de programmation. Pour cela nous utiliserons CAML qui permet de définir très simplement des syntaxes abstraites et aussi de construire des analyseurs. Ainsi, CAML jouera le rôle de "metalangage" par rapport aux langages manipulés qui pourront être C, JAVA ... ou CAML lui-même.
Les analyseurs seront implémentées à l'aide des outils OCAMLLEX et OCAMLYACC qui sont des adaptations à OBJECTIVE CAML de LEX et YACC et que nous décrivons succinctement ci-dessous.

3.1 Description de OCAMLLEX

OCAMLLEX est un utilitaire qui prend en entrée un fichier `lexer.mll` contenant un certain nombre de définitions d'unités lexicales et produit en sortie un fichier OBJECTIVE CAML `lexer.ml` contenant une définition de fonction par unité lexicale. Habituellement OCAMLLEX sert plutôt à réaliser des analyseurs lexicaux mais on peut aussi s'en servir pour réaliser d'autres fonctions sur les textes.

Par exemple, un fichier `lexer.mll` permettant de reconnaître un nombre flottant pourra contenir les définitions :

```
let chiffre = ['0'-'9']
let naturel = chiffre+
let signe = ['+' '-']
let entier = signe? naturel
let fraction  = '.' naturel
let exposant = ['e' 'E']  entier
```

```
let flottant = signe? ( (naturel? fraction exposant?)
                   |(naturel exposant)
                   |(naturel '.'))

rule flot =
  parse flottant     {float_of_string (Lexing.lexeme lexbuf)}
```

Les six premières lignes de ce fichier permettent de définir des abréviations pour des expressions régulières :
– `chiffre` désigne un chiffre décimal
– `naturel` désigne un nombre entier non signé
– `signe` désigne un signe
– `entier` désigne un entier relatif
– `fraction` désigne une partie fractionnaire d'un nombre flottant c'est-à-dire un point suivi d'un entier non signé
– `exposant` désigne une partie exponentielle d'un nombre flottant c'est-à-dire la lettre `"e"` ou `"E"` suivi d'un entier éventuellement signé

La définition suivante introduit léabréviation `flottant` pour les nombres flottants.

La suite du fichier définit une règle pour l'analyse des nombres flottants. Ce qui suit le mot-clé `rule` est le nom d'une unité lexicale (ici `flot`). Ce qui suit le mot-clé `parse` est une règle formée d'une expression régulière et d'une action sémantique (entre accolades). Cette action sémantique est une expressions quelconque de OBJECTIVE CAML. Toutefois, ces actions utilisent en général des fonctions du module `Lexing`.

En effet, les fonctions d'analyse lexicale produites par OCAMLLEX sont de type `Lexing.lexbuf -> ty` où `ty` est le type que doit retourner l'analyseur.

par exemple, ici, la commande

```
ocamllex lexer.mll
```

produit un fichier `lexer.ml` qui définit la fonction :

```
flot :  Lexing.lexbuf -> float
```

Le type `lexbuf` du module `Lexing` est le type des tampons d'analyse lexicale tel qu'il est défini dans le module `Lexing`. Ce module contient aussi la fonction `lexeme` qui lit le contenu du tampon et le retourne sous forme d'une chaîne de caractères. La fonction `float_of_string` transforme ensuite dans notre action sémantique cette chaîne de caractères en nombre flottant.

Le module `Lexing` contient aussi des fonctions permettant de construire des tampons d'analyse lexicale qui opèrent sur des chaînes de caractères ou sur un canal d'entrée. Ce sont :

```
Lexing.from_string : string -> lexbuf
Lexing.from_channel : in_channel -> lexbuf
```

Par exemple, pour lire un nombre flottant à partir du clavier, on pourra définir la fonction

```
let read_float () =
  let lexbuf = Lexing.from_channel stdin
  in flot lexbuf;;
```

puis utiliser cette fonction de la façon suivante

```
# read_float ();;
1.1E4
- : float = 11000
```

Cet analyseur de nombres flottants peut être amélioré en autorisant par exemple la présence de blancs devant le nombre flottant lu. Pour cela, il suffit d'ajouter une règle à la définition de `floatnum` :

```
rule flot =
 parse   ' '    {flot lexbuf}
   | signe? ( (naturel? fraction exposant?)
                   |(naturel exposant)
                   |(naturel '.'))
    {float_of_string (Lexing.lexeme lexbuf)}
```

L'action sémantique correspondant à la lecture d'un blanc est un appel récursif de la fonction d'analyse dont le paramètre, qui a pour type `Lexing.lexbuf`, s'appelle aussi par covention `lexbuf`.

En général, les analyseurs lexicaux produisent des valeurs d'un type appelé **token** utilisé par les analyseurs syntaxiques produits par OCAMLYACC. Par exemple, dans la section suivante, nous utiliserons un type `token` défini par :

```
type token =  PARG | PARD | VIRG | FLOAT of float
```

pour lequel l'analyseur lexical peut être décrit par

```
rule token =
 parse   ' '    {token lexbuf}
   |  '('       {PARG}
   |  ')'       {PARD}
   |  ','       {VIRG}
   |  flottant  {FLOAT (float_of_string (Lexing.lexeme lexbuf))}
```

A partir de cette description, OCAMLLEX produira une definition pour la fonction

```
token : Lexing.lexbuf -> token
```

3.2 Description de OCAMLYACC

Pour décrire OCAMLYACC, nous prendons d'abord l'exemple très simple de la lecture d'arbres binaires contenant des nombres flottants aux feuilles.

3.2.1 Analyse d'arbres

Le type CAML pour de tels arbres est :

```
type arbre = F of float | Bin of arbre * arbre
```

La syntaxe concrête de tels arbres sera écrite à l'aide de parenthèses et de virgules. Par exemple, l'écriture :

```
(2.3,(3.4,75E-1))
```

correspondra à la valeur

```
Bin(F 2.3,Bin(F 3.4,F 7.5))
```

On supposera que le type arbre est défini dans le module Arbre. Le fichier OCAMLYACC permettant la lecture des arbres binaires s'écrira :

```
(* Fichier parser.mly *)
%{
open Arbre
%}
%token PARG PARD VIRG
%token <float> FLOAT
%start arbre
%type <Arbre.arbre> arbre
%%
arbre :
   FLOAT                              {F $1}
 | PARG arbre VIRG arbre PARD   {Bin($2,$4)}
;
```

La section située entre `%{` et `%}` est un préambule permettant d'ouvrir des modules et éventuellement de définir des valeurs CAML utiles pour les actions sémantiques. Ensuite, on déclare les lexèmes utilisés qui sont ici `PARG`, `PARD` , `VIRG` et `FLOAT` auquel est attaché une valeur de type float. Le système OCAMLYACC utilise ces déclarations pour produire automatiquement une définition d'un type CAML `token` comme celui que nous avons écrit précédemment.
On déclare aussi `arbre` comme le point d'entrée principal de la grammaire et on donne le type produit par les actions sémantiques correspondant à ce point d'entrée. Enfin, après `%%`, on donne les règles pour le point d'entrée `arbre`.

Si cette définition de grammaire est placés dans un fichier `parser.mly`, la commande

```
ocamlyacc parser.mly
```

produit deux fichiers `parser.mli` et `parser.ml`. Le fichier `parser.mli` contient la définition du type `token` produite automatiquement et le fichier `parser.ml` contient la définition d'une fonction d'analyse correspondant au point d'entrée `arbre`.

```
arbre :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> Arbre.arbre
```

Cette fonction d'analyse prend en arguments un analyseur lexical et un tampon et produit un arbre. Pour l'utiliser, il nous faut encore définir avec OCAMLLEX l'anayseur lexical correspondant. Voici sa définition :

```
(* Fichier lexer.mll *)
{
open Parser
}

rule token =
 parse   ' '    {token lexbuf}
   |     '('    {PARG}
   |     ')'    {PARD}
   |     ','    {VIRG}
   |  flottant  {FLOAT (float_of_string (Lexing.lexeme lexbuf))}
```

La partie entre accolades est un préambule permettant d'ouvrir le module `Parser` où est défini le type `token`. Si ce texte est placé dans un fichier `lexer.mll`, la commande `ocamllex lexer.mll` produira un fichier `lexer.ml` qui contiendra la fonction :

```
token:  Lexing.lexbuf -> token
```

Pour utiliser l'analyseur syntaxique d'arbres à partir du clavier par exemple, il suffit maintenant de définir la fonction :

```
#let analyser_arbre () =
  let lexbuf = Lexing.from_channel stdin
  in arbre token lexbuf;;
# analyser_arbre();;
(2.3,(3.4,75E-1))
- : Arbre.arbre = Bin(F 2.3,Bin(F 3.4,F 7.5))
```

Et pour utiliser l'analyseur sur un fichier :

```
#let analyser_arbre_fichier  nom =
  let ch = open_in nom in
  let lexbuf = Lexing.from_channel ch
  in arbre token lexbuf;;
```

3.2.2 Ordre de compilation

On suppose qu'on a écrit 4 fichiers :
- `arbre.ml` qui contient la définition du type `arbre`
- `lexer.mll` qui contient la définition de l'analyseur lexical
- `parser.mly` qui contient la définition de l'analyseur syntaxique
- `analyse.ml` qui contient les fonctions dérivées comme `analyser_arbre`

L'ordre dans lequel doivent être compilés ces fichiers pour aboutir à des fonctions d'analyse utilisables est assez contraint :
- Pour lancer `ocamllex lexer.mll` , il faut que le module `Parser` existe et donc que le fichier `parser.cmi` existe. Pour cela, il faut que le fichier `parser.mli` existe et donc que la compilation " `ocamlyacc parser.mly` " ait été faite.
- Pour lancer `ocamlyacc parser.mly` , il faut que le module `Arbre` existe et donc que le fichier `arbre.cmi` existe.

La suite d'ordres de compilation peut être :

```
ocamlc -c arbre.ml          (* ->    arbre.cmi , arbre.cmo *)
ocamlyacc parser.mly        (* ->    parser.mli , parser.ml *)
ocamlc parser.mli           (* ->    parser.cmi  *)
ocamllex lexer.mll          (* ->    lexer.ml  *)
ocamlc -c lexer.ml          (* ->    lexer.cmi , lexer.cmo *)
ocamlc -c parser.ml         (* ->    parser.cmi , parser.cmo *)
ocamlc -c analyse.ml        (* ->    analyse.cmi , analyse.cmo *)
```

Enfin, on peut réunir les fichiers `.cmo` dans une bibliothèque :

```
ocamlc -a -o arbre.cma arbre.cmo lexer.cmo parser.cmo analyse.cmo
```

Bien entendu, on utilisera un `Makefile` pour déclencher automatiquement cette suite d'ordres :

```
#Makefile
OBJS = arbre.cmo lexer.cmo parser.cmo  analyse.cmo"

analyse.cma: $(OBJS)
                  ocamlc -a -o  analyse.cma $(OBJS)

lexer.ml: parser.cmi lexer.mll
                  ocamllex lexer.mll

.SUFFIXES: .ml .mli .mly .cmo .cmi

.ml.cmo:
                  ocamlc -c $<
.mli.cmi:|
                  ocamlc -c $<
.mly.mli:|
                  ocamlyacc -v $<
.mly.ml:|
                  ocamlyacc -v $<|
```

3.2.3 Analyse d'expressions arithmétiques

L'analyse d'expressions a rithmétiques fournit un exemple plus intéressant d'utilisation de YACC. D'une part, ces expressions apparaissent dans tous les langages de programmation. D'autre part, elles donne l'occasion de mettre en pratique les techniques de levée de l'ambiguitë vues à la section 2.6.4.

```
(* Fichier exp.ml *)
type exp = C of int
         | V of string
         | A of exp * exp
         | M of exp * exp


(* Fichier lexer.mll *)
{
  open Parser
}
```

```
let chiffre = ['0'-'9']
let entier = chiffre+
let lettre = ['a'- 'z' 'A'-'Z']
let  ident =(letter (lettre | chiffre|)*
let blanc = [' '  '\t' '\n']

rule token = parse
                  blanc          {token lexbuf}
         |        entier         {INT (int_of_string (Lexing.lexeme lexbuf))}
         |        ident          {IDENT  (Lexing.lexeme lexbuf)}
         |        '('            {PARG}
         |        ')'            {PARD}
         |        '+'            {PLUS}
         |        '*'            {MULT}


(* Fichier parser.mly *)
%{
open Exp
%}
%token <string> IDENT
%token <int> INT
%token PLUS MULT
%token PARG PARD
%left  PLUS
%left  MULT
%start expression
%type <Exp.exp> expression
%%
expression:
  IDENT                             {V($1)}
|  INT                              {C ($1)}
| expression PLUS expression        {A ($1, $3)}
| expression MULT expression        {M($1, $3)}
| PARG expression PARD              {$2}
;
```

Dans ce dernier fichier, les lignes qui permettent de désambiguer la grammaire sont

```
%left  PLUS
%left  MULT
```

Toutes deux indiquent que les opérateurs associent à gauche et par ailleurs, l'ordre dans lequel elles apparaissent indique l'ordre de priorités croissantes. Les opérateurs de priorités égales doivent apparaître sur une même ligne :

```
%left  PLUS MOINS
%left  MULT DIV
```

Certains symboles associent à droite comme par exemple le symbole `**` qui est traditionnellement utilisé pour la fonction exponentielle. Si le token `EXPO` correspond à ce symbole, on déclarera :

```
%right EXPO
```

Certains symboles ne peuvent pas être répétés sur un même niveau. Dans ce cas on les déclarent non-associatifs. Par exemple, on peut vouloir interdire une écriture telle que `x==y==z`. Dans ce cas, si le token `EGAL` correspond au symbole `==`, on déclarera :

```
%nonassoc EGAL
```

Enfin, certains symboles posent un problème particulier car sont utilisés avec plusieurs priorités. C'est le cas par exemple du symbone `-` qui peut être utilisé comme opérateur binaire de soustraction ou comme opérateur unaire d'opposé. Dans ce cas, la solution offerte par YACC consiste à déclarer un token fictif (par exemple `UMOINS`). On donne la priorité voulue à cet opérateur :

```
%nonassoc EGAL
%left  PLUS MOINS
%left  MULT DIV
%right EXPO
%nonassoc UMOINS
```

Puis, lorsque utilise, dans une règle de grammaire, le token `MOINS` comme opération unaire, on dispose d'une construction spéciale (`%prec`) pour donner à cette régle la priorité de `UMOINS`.

```
| MOINS expression    %prec UMOINS  {UMoins $2}
```

3.3 Définition de C en CAML

Une définition de syntaxe abstraite peut s'écrire de façon presque immédiate sous la forme de définitions de type CAML. Chaque catégorie syntaxique correspond à un type CAML et chaque constructeur de syntaxe abstraite correspond à un constructeur CAML. Voici une définition de la syntaxe abstraite de d'un sous-ensemble de C.

```
type cprog = Prog of cdecl list
and cdecl = VarDecl of  cvardecl
          | FunDecl of  cfundecl
and cvardecl = {varname: string; vartype: ctype}
and cfundecl = {funname: string; funrestype: ctype;
                   funparamstypes: (string * ctype) list;
                   funbody: cstat}
and cstat = Block of cdecl list * cstat list
          | Comput of cexp
          | If of cexp * cstat * cstat
          | For of cexp * cexp * cexp * cstat
          | Return of cexp
          | Nop
and cexp = Constexp of cconst
         | Nothing
         | Varexp of string
         | Unopexp of cunop * cexp
         | Binopexp of cbinop * cexp * cexp
```

```
         | Index of cexp * cexp
         | Indirection of cexp
         | Assign of cexp * cexp
         | Call of string * (cexp list)
and ctype = Voidtype | Inttype
           | Arraytype  of ctype | Pointertype of ctype
           | Funtype of ctype *  ctype list
and cconst = Int of int
and cunop = UMOINS | NOT
and cbinop = PLUS | MOINS | MULT | DIV
           | EGAL | LT | GT | LE | GE;;
```

On supposera qu'on dispose d'une fonction d'analyse :

`parse_file: string -> cprog`

qui lit le contenu d'un fichier C et produit la syntaxe abstraite correspondante.
Par exemple, supposons que le fichier "essai.c" contient les définitions suivantes :

```
int fact(int n)
{
 int i, m;
 m=1;
 for (i=2;i<n+1;i=i+1)    m=m*i;
 return(m);
   }

int fact2(int n)
{
 return(2*fact(n));
}

int facrec(int n)
{
 if (n==0) return(1);
  else return(n*facrec(n-1));
}
```

L'expression `parse_file "essai.c"` produira la syntaxe abstraite :

```
Prog
 [FunDecl
   {funname="fact"; funrestype=Cinttype; funparamstypes=["n", Cinttype];
    funbody=
     Block
      ([VarDecl {varname="i"; vartype=Cinttype};
        VarDecl {varname="m"; vartype=Cinttype}],
       [Comput (CAssign (Varexp "m", Constexp (Int 1)));
        For
         (Assign (Varexp "i", Constexp (Int 2)),
          Binopexp
           (LT, CVarexp "i",
```

```
           Binopexp (PLUS, Varexp "n", Constexp (Int 1))),
         Assign
          (Varexp "i", Binopexp (PLUS, Varexp "i", Constexp (Int 1))),
         Comput
          (Assign
            (Varexp "m", Binopexp (MULT, Varexp "m", Varexp "i"))));
       CReturn (CVarexp "m")])};
FunDecl
 {funname="fact2"; funrestype=Cinttype; funparamstypes=["n", Cinttype];
  funbody=
   Block
    ([],
     [Return
       (Binopexp
         (MULT, Constexp (Int 2), Call ("fact", [Varexp "n"])))])};
FunDecl
 {funname="facrec"; funrestype=Inttype; funparamstypes=["n", Inttype];
  funbody=
   Block
    ([],
     [If
       (Binopexp (EGAL, Varexp "n", Constexp (Int 0)),
        Return (Constexp (Int 1)),
        Return
         (Binopexp
           (MULT, Varexp "n",
            Call
             ("facrec",
              [Binopexp (MOINS, Varexp "n", Constexp (Int 1))])))])}]
```

3.4 Définition de JAVA en CAML

Nous présentons ci-dessous une version simplifiée mais relativement étendue malgré tout du langage JAVA. Nous avons supprimé tout ce qui concerne les packages et aussi les modifieurs associés aux classes et aux interfaces. Les classes et les interfaces seront considérées comme publiques et non finales. Nous avons supprimé également les déclarations de constructeurs. On utilisera uniquement les constructeurs par défaut. Les types de base sont restreints pour simplifier à `void`, `boolean` et `int` et il n'y a pas de casts.

On commence par la définition de quelques types utilitaires. Le type `status` permettra de marquer un champ ou une méthode introduite dans une classe comme étant attaché aux instances ou à la classe (mot-clé `static`).

```
type  'a option = Nopt | Some of 'a;;
type   protection = Private | Protected | Public;;
type   status = InstStatus | ClassStatus;;
```

Classes et interfaces

Un programme JAVA sera considéré comme une liste de déclarations de classes et d'interfaces. Dans la syntaxe concrête, ces deux types de déclarations peuvent être mis dans un ordre quelconque mais dans la syntaxe abstraite, on regroupe par commodité toutes les déclarations de classes d'un côté et toutes les déclarations d'interfaces de l'autre.
Une déclaration d'interface comporte son nom, la liste des noms des interfaces dont elle hérite et la liste de ses en-têtes de méthodes. Une déclarations de classe comporte son nom, le nom de son père, la liste des noms des interfaces dont elle hérite, la liste de ses champs et la liste de ses méthodes.

```
type javaprog = JavaProg of (javainterdecl list) * (javaclassdecl list)


and javainterdecl = {internname: string;
                     interfather: string list;
                     methheads: javamethdecl list}


and javaclassdecl = {classname: string;
                     classfather: string option;
                     classinters: string list;
                     fielddecls: javafielddecl list;
                     methdecls: javamethdecl list}
```

Déclaration de méthodes et de champs

```
and javamethdecl = {methname: string;
                    methrestype: javatype;
                    methparams: (string * javatype) list;
                    methprotection: protection;
                    methstatus: status;
                    methexns: javaident list;
                    methbody: javastat option}


and javafielddecl = {fieldname: string;
                     fieldtype: javatype;
                     fieldprotection: protection;
                     fielstatus: status;
                     fieldinit: javaexp option}

and javavardecl = {varname: string;
                   vartype: javatype;
                   varinit: javaexp option}

and javatype = JavaVoidType
             | JavaBoolType | JavaIntType
             | JavaIdType of javaident
             | JavaArrayType of javatype
```

```
and javaident = string
```

Expressions

```
and javaexp =  JavaConstExp of javaconst
             | JavaThisExp
             | JavaVarExp of javaident
             | JavaNewClassExp of javaident
             | JavaNewArrayExp of javatype * (javaexp list)
             | JavaMethCall of javaexp * string * (javaexp list)
             | JavaCall of string *  (javaexp list)
             | JavaCallSuper of string * (javaexp list)
             | JavaFieldExp of javaexp * string
   | JavaFieldSupExp of string
             | JavaIndex of javaexp * javaexp
             | JavaUnopExp of javaunop * javaexp
             | JavaBinopExp of javabinop * javaexp * javaexp
             | JavaAssign of javaexp * javaexp

and javaconst = JavaInt of int
              | JavaTrue
              | JavaFalse
              | JavaNull

and javaunop = JavaUMOINS | JavaNOT
and javabinop = JavaAND | JavaOR | JavaPLUS | JavaMOINS | JavaMULT | JavaDIV
              | JavaEGAL | JavaNE | JavaLT | JavaGT | JavaLE | JavaGE
```

Blocks et instructions

```
and javastat = JavaBlock of (javavardecl list) * (javastat list)
          | JavaComput of javaexp
          | JavaIf of javaexp * javastat * javastat
          | JavaFor of javaexp * javaexp * javaexp * javastat
          | JavaReturn of javaexp option
          | JavaTry of javastat * (javacatch list)
          | JavaThrow of javaexp
          | JavaNop

and javacatch = {eid: string;
 etype: javatype;
                 stat: javastat}
;;
```

3.5 Définition de CAML en CAML

```
type exp =
    Int of int                          (*  constante entiere *)
```

```
| Bool of bool                      (*  constante booleenne *)
| Pair of exp * exp                 (*  paire *)
| Op of string
| Var  of string                    (* variable *)
| If  of exp * exp * exp            (* conditionnelle *)
| Fun  of string * exp              (* fonction *)
| App  of exp * exp                 (* application *)
| Let of string * exp * exp         (* declaration *)
| Letrec of string * exp * exp      (* declaration recursive *)
```

Chapitre 4

Typage

4.1 Jugements et règles de typage

La définition précise du typage correct d'un langage de programmation exprime par des règles la façon dont le typage d'un élément de programme dépend du typage de ses constituants. Elle suppose donc qu'un élément de programme soit représenté d'une façon qui met en évidence sa structure et c'est précisément ce que fournit la notion de syntaxe abstraite.
Un programme donné sous forme de syntaxe abstraite est une expression formée d'un constructeur et d'arguments qui sont eux-mêmes des expressions. Le typage est donc défini de façon naturelle en donnant une règle de typage par constructeur de syntaxe abstraite. Dans l'écriture des règles, on s'autorisera cependant à écrire les éléments de programme sous forme de syntaxe concrête pour plus de lisibilité mais il est important de garder en mémoire que les règles de typage sont en correspondance avec les constructeurs de syntaxe abstraite.
Le typage d'un élément de programme comportant des variables libres, c'est-à-dire des variables qui n'ont pas de lieurs dans le morceau de programme considéré, va dépendre des types attribués à ces variables. On appellera **environnement de types** une structure ou une fonction définissant les types d'un certain nombre de variables. On n'établira pas en général le typage d'un élément de programme dans l'absolu mais relativement à un environnement de types donné.
Par exemple, le typage d'une expression e sera donné sous la forme d'un **jugement de typage** de la forme :

$$E \vdash e : t$$

qui se lira :

"Dans l'environnement de types E, l'expression e a le type t ".

Lorsque l'élément de programme à analyser est une déclaration, celle-ci ne possède pas de type à proprement parler mais elle augmente l'environnement de types. Le typage d'une déclaration d sera donc défini par un jugement de la forme :

$$E \vdash d : E'$$

qui se lira :

"Dans l'environnement de types E, la déclation d produit un nouvel environnement de types E' ".

Les environnements de types peuvent être représentés de différentes façons. Le plus simple est de considérer qu'il s'agit de fonctions partielles dont l'argument est un identificateur et le résultat est le type de cet identificateur. C'est le point de vue que nous adopterons pour la définition des règles de typage ci-dessous. Par contre, qund nous implanterons ces règles de typage, nous pourrons choisir d'autres représentations (par exemple des listes d'association).
Quelle que soit la représentation, il sera nécessaire de pouvoir définir facilement des environnements simples et de composer des environnements.
Nous noterons [] l'environnement vide c'est-à-dire la fonction indéfinie partout. Nous noterons $[x, t]$ l'environnement qui associe le type t à l'identificateur x est qui est indéfini pour tout autre identificateur. Par extension, nous noterons $[x_1, t_1; ...; x_n, t_n]$ l'environnement qui associe pour tout i compris entre 1 et n, le type t_i à l'identificateur x_i et qui est indéfini pour tout autre identificateur. Enfin, si E_1 et E_2 sont deux environnements, nous noterons $E_1 \oplus E_2$ l'environnement E' défini par

$$E'(x) = \begin{cases} E_1(x) \text{ si } E_1(x) \text{ existe} \\ E_2(x) \text{ sinon} \end{cases}$$

Une règle de typage définit le typage d'un élément de programme à partir du typage de ses constituants directs. Elle se présente sous la forme d'une règle de déduction écrite comme une fraction. Ce qui est écrit sous la barre de fraction est le jugement de typage qui constitue la conclusion de la règle. Ce qui est écrit au-dessus de la barre est l'ensemble des hypothèses de la règle. Ces hypothèses sont soit des jugements de typage concernant les constituants de l'élément de programme à typer soit des conditions supplémentaires apparaissant sous la forme d'égalités.

$$\frac{Hyp_1 \quad ... \quad Hyp_n}{Conclusion} \text{ (Regle)}$$

4.2 Typage d'un langage d'expressions

Nous allons donner des règles de typage pour un petit langage dont la syntaxe ressemble à celle de CAML. On part d'un langage permettant d'écrire des expresions arithmétiques et booléennee en considérant uniquement les types entier et booléen puis on ajoute la construction `let ... in ...`, puis les paires et enfin les fonctions et leur application. Contrairement à ce qui se passe en CAML., nous ne ferons pas ici de synthèse de type mais seulement de la vérification : ceci suppose que l'utilisateur indique le type du paramètre d'une fonction.

4.2.1 Première partie du langage

```
type exp =
    Var of string
  | Const of int
  | Vrai   |  Faux
  | Non of exp  | Ou of exp * exp   | Et of exp * exp
  | Egal of exp * exp   | Diff of exp * exp
  | Plus of exp * exp   | Moins of exp * exp
  | Mult of exp * exp   | Div of exp * exp
```

```
  | If of exp * exp *exp
  | Let of string * exp * exp
and tyexp =    TyInt   |  TyBool
```

Les seuls types considérés pour le moment sont `int` et `bool`. Ils ne font pas partie pour le moment de la syntaxe du langage car, comme il n'y a pas pour le moment de déclaration,on peut se dispenser de décrire la syntaxe abstraite des types. Ce sera différents un peu plus loin.

Les premières régles donnent le type des constantes :

$$\frac{}{\text{E} \vdash \text{Const } n : \text{ TyInt}} \text{ (Const)}$$

$$\frac{}{\text{E} \vdash \text{Vrai} : \text{ TyBool}} \text{ (Vrai)} \qquad \frac{}{\text{E} \vdash \text{Faux} : \text{ TyBool}} \text{ (Faux)}$$

La règle concernant les variables est :

$$\frac{\text{E(x)=t}}{\text{E} \vdash \text{Var x} : \text{ t}} \text{ (Var)}$$

Ensuite viennent les règles correspondant aux différents opérat eurs.

$$\frac{\text{E} \vdash e : \text{ TyBool}}{\text{E} \vdash \text{Non(e)} : \text{ TyBool}} \text{ (Non)} \qquad \frac{\text{E} \vdash e_1 : \text{ TyBool} \quad \text{E} \vdash e_2 : \text{ TyBool}}{\text{E} \vdash \text{Ou}(e_1, e_2) : \text{ TyBool}} \text{ (Ou)}$$

$$\frac{\text{E} \vdash e_1 : \text{ t} \quad \text{E} \vdash e_2 : \text{ t}}{\text{E} \vdash \text{Egal}(e_1, e_2) : \text{ TyBool}} \text{ (Egal)} \qquad \frac{\text{E} \vdash e_1 : \text{ TyInt} \quad \text{E} \vdash e_2 : \text{ TyInt}}{\text{E} \vdash \text{Plus}(e_1, e_2) : \text{ TyInt}} \text{ (Plus)}$$

La règle de l'égalité introduit pour la première fois une variable de type (notée t) et cette variable apparaît 2 fois : comme type de e_1 et comme type de e_2. C'est une façon élégante de noter que les deux membres d'une égalité doivent avoir le même type. L'utilisation faite ici de cette variable t est exactement la même que celle qui est faite dans toutes les règles de la variable d'environnement E : dans une règle, toutes les occurrences de E sont supposées représenter le même environnement. E est une variable qui représente n'importe quel environnement mais les différentes occurences de E dans une même règle doivent être instanciées de la même façon.

Pour plus de lisibilité, on s'autorise souvent à utiliser la syntaxe concrête plutôt que la syntaxe abstraite dans l'écriture des règles. Oar exemple, la règle (Ou) pourra aussi s'écrire :

$$\frac{\text{E} \vdash e_1 : \text{ bool} \quad \text{E} \vdash e_2 : \text{ bool}}{\text{E} \vdash e_1 \parallel e_2 : \text{ bool}} \text{ (Ou)}$$

Nous continuons avec ce type d'écriture pour les règles suivantes.

$$\frac{\text{E} \vdash \text{e} : \text{ bool} \quad \text{E} \vdash \text{e}_1 : \text{ t} \quad \text{E} \vdash \text{e}_2 : \text{ t}}{\text{E} \vdash \text{if e then e}_1 \text{ in e}_2 : \text{ t}} \text{ (If)}$$

Enfin, la règle du `let`, du fait qu'elle introduit une nouvelle variable, comporte un accroissement de l'environnement :

$$\frac{\text{E} \vdash \text{e}_1 : \ t_1 \quad [x : t_1] \oplus \text{E} \vdash \text{e}_2 : \ t_2}{\text{E} \vdash \text{let x} = \text{e}_1 \text{ in e}_2 : \ t_2} \text{ (Let)}$$

4.2.2 Deuxième partie du langage

On enrichit maintenant le langage avec des constructions qui le rapproche de CAML : les paires et les fonctions. L'utilisateur doit cependant déclarer le type du paramètre dans une déclaration de fonction, ce qui oblige à ajouter les types dans la syntaxe abstraite.

```
type exp =
    ...
 | Fst of exp  | Snd of exp
 | Paire of exp * exp71
 | Fonct of string * tyexp * exp
 | App of exp * exp

and tyexp =
    ...
 |  TyPaire of tyexp * tyexp
 |  TyFleche of tyexp * tyexp
```

Si non avons déclaré `Fst` et `Snd` comme des opérateurs à un argument et non comme des constantes, c'est pour ne pas avoir à leur donner un type, ce qui n'est possible qu'en introduisant du polymorphisme.

$$\frac{\mathrm{E} \vdash e : \ t_1 * t_2}{\mathrm{E} \vdash Fst(e) : \ \mathrm{t1}} \ (\mathrm{Fst}) \qquad \frac{\mathrm{E} \vdash e_1 : \ t_1 \quad \mathrm{E} \vdash e_2 : \ t_2}{\mathrm{E} \vdash e_1, e_2 : \ t_1 * t_2} \ (\mathrm{Paire})$$

La règle permettant de typer une définition de fonction s'écrit :

$$\frac{[x : t] \oplus \mathrm{E} \vdash \mathrm{e} : \ t'}{\mathrm{E} \vdash \text{function (x :t)} \rightarrow \mathrm{e} \ : \ t \rightarrow t'} \ (\mathrm{Fonct})$$

La règle permettant de typer une application de fonction s'écrit :

$$\frac{\mathrm{E} \vdash \mathrm{f} : \ t_1 \rightarrow t_2 \quad \mathrm{E} \vdash \mathrm{e} : \ t_1}{\mathrm{E} \vdash \mathrm{f\ e} : \ t_2} \ (\mathrm{App})$$

4.2.3 Exemple

Si on se place dans l'environnement

`E = [f: bool -> bool; x:bool; y:bool]`

on peut démontrer à l'aide des règles précédentes le jugement $\mathrm{E} \vdash \mathrm{f(x} \parallel \mathrm{y)} : \ \mathrm{bool}$

En voici la preuve :

$$\frac{\dfrac{\mathrm{E} \vdash e_1 : \ \mathrm{bool} \quad \mathrm{E} \vdash e_2 : \ \mathrm{bool}}{\mathrm{E} \vdash e_1 \parallel e_2 : \ \mathrm{bool}} \ (\mathrm{Ou}) \qquad \mathrm{E} \vdash \mathrm{f} : \ \mathrm{bool} \rightarrow \mathrm{bool}}{\mathrm{E} \vdash \mathrm{f(x} \parallel \mathrm{y)} : \ \mathrm{bool}} \ (\mathrm{App})$$

4.2.4 Un vérificateur de types

```
exception ErreurDeTypage of string;;

let rec typerExp env e =
match e
with Var x -> (try List.assoc x env
               with Not_found -> raise (ErreurDeTypage
                                        ("variable "
                                        ^ x ^ " non definie")))
|   Const _ -> TyInt
|   Vrai  -> TyBool
|   Faux  -> TyBool
|   Non e  -> (match typerExp env e
              with TyBool -> TyBool
               | _ -> raise (ErreurDeTypage ("Non: mauvais argument")))
|  Ou(e1,e2)  -> (match (typerExp env e1, typerExp env e2)
                  with TyBool,TyBool  -> TyBool
                | _ -> raise (ErreurDeTypage ("Ou: argument non booleen")))
|  Et(e1,e2)  -> (match (typerExp env e1, typerExp env e2)
                  with TyBool,TyBool  -> TyBool
                | _ -> raise (ErreurDeTypage ("Et: argument non boolee")))
|  Egal(e1,e2)  -> if typerExp env e1 = typerExp env e2
                  then   TyBool
                  else  raise (ErreurDeTypage ("Egal: arguments de types differents"))
|  Diff(e1,e2)  -> if typerExp env e1 = typerExp env e2
                  then   TyBool
                  else  raise (ErreurDeTypage ("Egal: arguments de types differents"))

|  Plus(e1,e2)  -> (match (typerExp env e1, typerExp env e2)
                   with TyInt,TyInt  -> TyInt
                | _ -> raise (ErreurDeTypage ("Add: argument non entier")))
|  Moins(e1,e2)  -> (match (typerExp env e1, typerExp env e2)
                   with TyInt,TyInt  -> TyInt
                | _ -> raise (ErreurDeTypage ("Moins: argument non entier")))
|  Mult(e1,e2)  -> (match (typerExp env e1, typerExp env e2)
                   with TyInt,TyInt  -> TyInt
                | _ -> raise (ErreurDeTypage ("Mult: argument non entier")))
|  Div(e1,e2)  -> (match (typerExp env e1, typerExp env e2)
                   with TyInt,TyInt  -> TyInt
                | _ -> raise (ErreurDeTypage ("Div: argument non entier")))

|  If(e,e1,e2) ->  if typerExp env e <> TyBool
                      then   raise (ErreurDeTypage ("If: test non booleen"))
                      else let t= typerExp env e1 and t'= typerExp env e2
                            in if t=t' then t
                               else raise (ErreurDeTypage ("If: alternants de types differents")

| Let(x,e1,e2)
           -> let t= typerExp env e1 in typerExp ((x,t)::env) e2
```

```
| Fst e ->  (match typerExp env e
           with TyPaire(t1,t2)  -> t1
             |      _   -> raise (ErreurDeTypage ("Fst: l'argument n'est pas une paire

| Snd e ->  (match typerExp env e
           with TyPaire(t1,t2)  -> t2
             |      _   -> raise (ErreurDeTypage ("Snd: l'argument n'est pas une paire

| Paire(e1,e2) -> TyPaire(typerExp env e1,typerExp env e2)

| Fonct(x,t,e) -> TyFleche(t,typerExp ((x,t)::env) e)

| App(e1,e2) -> (match (typerExp env e1,typerExp env e2)
                 with TyFleche(t1,t2), t1' ->
                      if t1=t1' then t2 else  raise (ErreurDeTypage ("App: l'argument n
                  |  _  ->  raise (ErreurDeTypage ("App: application d'une valeur non-f
;;
```

4.3 Le typage de C

4.3.1 Les types de C

Nous nous limiterons ici aux deux types de base `int` et `void`, ce qui nous permettra d'éviter d'avoir à traiter la surcharge des opérateurs arithmétiques.
Nous considérerons par ailleurs les constructeurs `array` et `pointer` et, pour les fonctions, nous noterons $(t_1 * \ldots * t_n \rightarrow t)$ le type d'une fonction ayant des arguments de types $t_1, \ldots, t_n$ et un résultat de type t. Une fonction sans argument dont le résultat est de type t aura pour type $(void \rightarrow t)$.

4.3.2 Règles pour le typage de C

La règle la plus simple est celle qui concerne les variables. Etant donné un environnement de types E, on désignera par E(x) le type de la variable x dans l'environnement E. La règle de typage d'une variable s'écrit donc :

$$\frac{\mathrm{E(x)=t}}{\mathrm{E} \vdash \mathrm{x} : \mathrm{t}} \text{ (Var)}$$

En ce qui concerne les expressions consistant en l'application d'un opérateur unaire ou binaire à un ou deux arguments, on notera $(t_1 \rightarrow t)$ et $(t_1 * t_2 \rightarrow t)$ les types des opérateurs. Ces types sont supposés associés à la définition du langage.

Les règles sont les suivantes

$$\frac{\mathrm{E} \vdash \mathrm{e} : \mathrm{t1} \qquad \mathrm{op} : t_1 \rightarrow t}{\mathrm{E} \vdash \mathrm{op\ e} : \mathrm{t}} \text{ (Unop)}$$

$$\frac{\mathrm{E} \vdash e_1 : t_1 \qquad \mathrm{E} \vdash e_2 : t_2 \qquad \mathrm{op} : t_1 * t_2 \rightarrow t}{\mathrm{E} \vdash e_1\ \mathrm{op}\ e_2 : \mathrm{t}} \text{ (Binop)}$$

La règle concernant l'indexation dans un tableau s'écrit :

$$\frac{\mathrm{E} \vdash e_1 : \ \mathrm{array(t)} \qquad \mathrm{E} \vdash e_2 : \ \mathrm{int}}{\mathrm{E} \vdash e_1[e_2] : \ \mathrm{t}} \ \text{(Index)}$$

La règle concernant l'accès à la valeur d'un pointeur s'écrit :

$$\frac{\mathrm{E} \vdash e : \ \mathrm{pointer(t)}}{\mathrm{E} \vdash *\mathrm{e} : \ \mathrm{t}} \ \text{(Indirection)}$$

La règle concernant la création d'un pointeur s'écrit :

$$\frac{\mathrm{E} \vdash \mathrm{e} : \ \mathrm{t} \qquad \mathrm{assignable(e)}}{\mathrm{E} \vdash \&e : \ \mathrm{pointer(t)}} \ \text{(Address)}$$

La règle concernant l'affectation s'écrit :

$$\frac{\mathrm{E} \vdash e_1 : \ t \qquad \mathrm{E} \vdash e_2 : \ t \qquad \mathrm{assignable(e)}}{\mathrm{E} \vdash e_1 = e_2 : \ \mathrm{t}} \ \text{(Assign)}$$

On peut noter que cette règle peut être plus complexe si on autorise les coercions dans les affectations. Ici, nous demandons que les expressions figurant à gauche et à droite d'un signe d'affectation aient exactement le même type. Par ailleurs, pour qu'une affectation ou l'application d'un opérateur d'adresse ait un sens, Il faut que l'expression qui se trouve à gauche du signe d'affectation ou celle à qui on applique l'opérateur d'adresse, désigne un emplacement mémoire, ce qui est noté ici par la propriété "assignable".

La règle concernant les appels de fonctions s'écrit :

$$\frac{\mathrm{E} \vdash f : \ t_1 * \ldots * t_n \ \rightarrow \ t \qquad \mathrm{E} \vdash e_i : \ t_i \quad 1 \leq i \leq n}{\mathrm{E} \vdash f(e_1, ..., e_n) : \ \mathrm{t}} \ \text{(Call)}$$

Ici encore, on pourrait assouplir la règle en utilisant des coercions sur le type des arguments.

La règle ci-dessus admet un cas particulier pour les fonctions sans arguments :

$$\frac{\mathrm{E} \vdash f : \ \mathrm{void} \ \rightarrow \ \mathrm{t}}{\mathrm{E} \vdash f() : \ \mathrm{t}} \ \text{(Call0)}$$

Nous avons terminé la liste des règles permettant de typer les expressions et nous poursuivons par les règles permettant de typer les instructions (statements). Les instructions n'ont pas de type en elles-mêmes. Simplement, elles sont correctement typées si les expressions qu'elles contiennent le sont. On exprimera donc la correction du typage des instructions par un jugement de la forme

$$\mathrm{E} \vdash \mathrm{s}$$

signifiant : dans l'environnement E, l'instruction s est bien typée.

Notons qu'une instruction en C peut être une expression. Dans ce cas, c'est-à-dire quand une expression est utilisée comme instruction, sa valeur est perdue. Il n'est donc utile d'utiliser une expression comme instruction que lorsque celle-ci a un effet, par exemple une affectation ou un appel de fonction qui réalise des opérations qui ne se limitent pas au calcul d'une expression.
La règle correspondante est :

$$\frac{\mathrm{E} \vdash \mathrm{e} : \ \mathrm{t}}{\mathrm{E} \vdash (e\,;)} \ (\mathrm{Exp\_Stat})$$

Typage de la conditionnelle :

$$\frac{\mathrm{E} \vdash e : \ \mathrm{int} \qquad \mathrm{E} \vdash s_1 \qquad \mathrm{E} \vdash s_2}{\mathrm{E} \vdash \mathrm{if}\ e\ s_1\ \mathrm{else}\ s_2} \ (\mathrm{If})$$

Typage de la boucle while :

$$\frac{\mathrm{E} \vdash \mathrm{e} : \ \mathrm{int} \qquad \mathrm{E} \vdash \mathrm{stat}}{\mathrm{E} \vdash \mathrm{while\ (e)\ stat}} \ (\mathrm{While})$$

Typage de la boucle for :

$$\frac{\mathrm{E} \vdash e_1 : \ t_1 \qquad \mathrm{E} \vdash e_2 : \ int \qquad \mathrm{E} \vdash e_3 : \ t_3 \qquad \mathrm{E} \vdash \mathrm{s}}{\mathrm{E} \vdash \mathrm{for}\ (e_1; e_2; e_3)\ \mathrm{s}} \ (\mathrm{For})$$

Le manuel de référence du langage C indique que l'exécution d'une boucle "for $(e_1; e_2; e_3)$ s" se passe de la façon suivante :

1. L'expression e_1 est évaluée pour ses effets (sa valeur est jetée).
2. L'expression e_2 est évaluée et si sa valeur est 0, la boucle est terminée, sinon, on passe à l'étape 3.
3. Le corps de la boucle s est exécuté.
4. L'expression e_3 est évaluée pour ses effets (sa valeur est jetée).
5. On retourne à l'étape 2.

Ceci implique que le type des expressions e_1 et e_3 est quelconque et que, dans la mesure où nous n'autorisons pas les conversions, le type de e_2 est nécessairement `int`. Dans le vrai langage C (avec les conversions), le type de e_2 est n'importe quel type convertible en entier. En particulier, ce peut être un type pointeur, la valeur 0 correspondant alors au pointeur vide.

Une suite d'instructions sera correcte si chacune des instructions de la suite est correcte. En particulier, une suite vide d'instructions est toujours correcte :

$$\frac{\mathrm{E} \vdash \mathrm{stat} \qquad \mathrm{E} \vdash \mathrm{stats}}{\mathrm{E} \vdash \mathrm{stat} \quad \mathrm{stats}} \ (\mathrm{Stats}) \qquad \frac{}{\mathrm{E} \vdash /^*\ \mathrm{vide}\ ^*/} \ (\mathrm{Stats0})$$

Enfin, il nous reste à définir le typage des déclarations. On considérera que le typage d'une déclaration dans un environnement E est un nouvel environnement E' obtenu en rajoutant à E ce qu'introduit cette déclaration.
La règle concernant les déclarations de variables est simple :

$$\frac{}{\mathrm{E} \vdash \mathrm{t\ x} : \ [x, t] \oplus \mathrm{E}} \ (\mathrm{VarDecl})$$

Le typage des déclarations de fonctions est un peu délicat car les fonctions peuvent être récursives. L'environnement servant à typer le corps de la fonction doit donc contenir le type de la fonction elle-même. Par ailleurs, les fonctions peuvent contenir différents points de sortie (intructions `return` qui doivent toutes renvoyer le même type résultat. Pour effectuer cette vérification, on introduira dans l'environnement servant à typer le corps de la fonction une variable spéciale `return` qui sera associée au type résultat :

$$\frac{[f, t_1 \times \ldots \times t_n \ \rightarrow \ t \ ; \ return, t] \oplus [x_1, t_1; ...; x_n, t_n] \oplus \mathrm{E} \vdash \mathrm{stat}}{\mathrm{E} \vdash \mathrm{t\ f}(t_1\ x_1, ..., t_n\ x_n)\ \mathrm{stat}: \ [f, (t_1 \times \ldots \times t_n \ \rightarrow \ t)] \oplus \mathrm{E}} \ (\mathrm{FunDecl})$$

La règle associée à la consruction `return` est alors :

$$\frac{\mathrm{E} \vdash \mathrm{e}: \ \mathrm{t} \qquad \mathrm{E(return)} = \mathrm{t}}{\mathrm{E} \vdash \mathrm{return(e)}\,;} \ (\mathrm{Return})$$

et, dans le cas particulier d'un return sans argument :

$$\frac{\mathrm{E(return)} = \mathrm{void}}{\mathrm{E} \vdash \mathrm{return}\,;} \ (\mathrm{Return0})$$

Il nous faut des règles pour traiter les suites de déclarations :

$$\frac{\mathrm{E} \vdash \mathrm{decl}: \ \mathrm{E'} \qquad \mathrm{E'} \vdash \mathrm{decls}: \ \mathrm{E''}}{\mathrm{E} \vdash \mathrm{decl} \quad \mathrm{decls}: \ \mathrm{E''}} \ (\mathrm{Decls}) \qquad \frac{}{\mathrm{E} \vdash /^*\ \mathrm{vide}\ ^*/ : \ \mathrm{E}} \ (\mathrm{Decls0})$$

Enfin, nous pouvons à présent typer les blocks, qui sont constitués d'une suite de déclarations et d'une suite d'isntructions :

$$\frac{\mathrm{E} \vdash \mathrm{decls}: \ \mathrm{E'} \qquad \mathrm{E'} \vdash \mathrm{stat}}{\mathrm{E} \vdash \{\mathrm{decls} \quad \mathrm{stats}\}} \ (\mathrm{Block})$$

4.3.3 Un vérificateur de types pour C en CAML

A partir de ces règles, il est facile de produire un vérificateur de types. Il suffit d'utiliser ces règles à l'envers.

```
exception Erreur_Typage of string;;
let library_env = [];;
let type_cunop cop = Cinttype,Cinttype;;
let type_cbinop bop = Cinttype,Cinttype,Cinttype;;

let eval e = ();;

let rec typer_cprog  (CProg (decls))
  = typer_cdecls library_env decls
and typer_cdecls env decls  = fold_left typer_cdecl env decls
and typer_cdecl env decl  =
 match decl
 with  CVarDecl {varname=v; vartype=t} ->  (v,t)::env
   |   CFunDecl   {funname=f; funrestype=t;
                   funparamstypes=vts; funbody= stat}
    ->   let env1 = (f,Cfuntype(t,vts)):: env in
         let env2 = vts@env1 in
         let t' = typer_cstat (("return",t)::env2) stat
         in if t'=t then env1
                    else raise (Erreur_Typage ("wrong type in fonction" ^ f))
and typer_cstat env stat =
  match stat
  with CBlock(decls,stats)
```

```
         ->  let env1 = typer_cdecls env decls
             in typer_cstats env1 stats
   |   CComput(exp) -> typer_cexp env exp
   |   CIf(e,st1,st2)
        -> let t= typer_cexp env e
           and t1 = typer_cstat env st1
           and t2 = typer_cstat env st2
           in if t=Cinttype && t1=t2 then true
                                    else raise (Erreur_Typage "if")
   |   CFor(e1,e2,e3,st)
        -> let t1 = typer_cexp env e1
           and t2 = typer_cexp env e2
           and t3 = typer_cexp env e3
           and t =  typer_cstat env st
           in if t1=Cinttype &&  t2=Cinttype &&  t3=Cinttype
                 then true
                 else raise (Erreur_Typage "for")
   |   CReturn(e) -> let t' = typer_cexp env e
                     and t'' = assoc "return" env
                     in if t''=t' then true
                           else  raise (Erreur_Typage "wrong return type")
   |   CNop ->  true
and typer_cstats env stats =
  match stats
  with [] ->  raise (Erreur_Typage "Pas de statement")
   | [st] -> typer_cstat env st

   | st::sts -> begin eval(typer_cstat env st) ;
                      typer_cstats env sts end



and typer_cexp env e =
 match e
 with CConstexp (CInt n) -> Cinttype
  |  CVarexp (v)
         ->( try assoc v env
            with _ ->raise (Failure ("Var " ^ v ^ " non definie")))
  |  CUnopexp (cunop,e1)
    ->  let (t1,t2) = type_cunop cunop
        and t = typer_cexp env e1
        in if t=t1 then t2
                   else raise (Erreur_Typage "wrong unop arg")
  | CBinopexp(cbinop,e1,e2)
    -> let (t1,t2,t) = type_cbinop cbinop
       and t1' = typer_cexp env e1
       and t2' = typer_cexp env e2
       in if t1=t1' && t2=t2' then t
                   else raise (Erreur_Typage "wrong binop arg")
```

```
  | CIndex(tab,idx)
    -> let t1 = typer_cexp env tab
       and t2 = typer_cexp env idx
       in (match t1
          with (Carraytype t)
              -> if t2=Cinttype then t
                    else  raise (Erreur_Typage "wrong index type")
            |   _ ->  raise (Erreur_Typage"wrong index type" ))
  |   CIndirection e
    -> (match typer_cexp env e
        with  Cpointertype t -> t
         | _ ->  raise (Erreur_Typage "wrong indirection" ))


  |   CAssign(e1,e2)
         ->  let t1 = typer_cexp env e1
             and t2 = typer_cexp env e2
             in if t1 = t2 then t1
                    else raise (Erreur_Typage "wrong assignment")

  |   CCall(f, args)
         -> (match (try assoc f env
                    with _ ->raise (Failure ("Var " ^ f ^ " non definie")))
             with Cfuntype(t,ts)
                -> let ts' = map (typer_cexp env) args
                   in if (map snd ts) =ts' then t
                                else raise (Erreur_Typage "wrong call arg")
              | _ ->  raise (Erreur_Typage "call to non function"))

and typer_cexps env es =
  match es
  with [] ->  raise (Erreur_Typage "Pas d'expression")
   | [e] -> typer_cexp env e
   | e::es -> begin eval (typer_cexp env e); typer_cexps env es end
;;


let typer_cdecls decls = rev(typer_cdecls [] decls) ;;
let typer_cdecl decl = hd (typer_cdecl [] decl);;
let typer_cprog p = rev(typer_cprog p);;
```

4.4 Règles de typage pour JAVA

4.4.1 Notation des types

Les types de JAVA considérés seront :
- void
- boolean
- int

– Inst(C) (le type correspondant à une classe C)

4.4.2 La relation de sous-classe

Le fait que la classe C est une sous-classe(au sens large) de la classe D dans le programme P s'écrira

$$\mathrm{P} \vdash \mathrm{C} \prec \mathrm{D}$$

Le fait que la classe C est une sous-classe immédiate de la classe D dans le programme P s'écrira

$$\mathrm{P} \vdash \mathrm{C} \text{ extends } \mathrm{D}$$

4.4.3 La relation de sous-type

Le fait que le type t_1 est un sous-type du type t_2 dans le programme P s'écrira

$$\mathrm{P} \vdash t_1 \leq t_2$$

Cette relation est définie par les règles suivantes :

$$\frac{}{\mathrm{P} \vdash \mathrm{t} \leq \mathrm{t}}$$

$$\frac{\mathrm{P} \vdash \mathrm{C} \prec \mathrm{D}}{\mathrm{P} \vdash \mathrm{Inst\ C} \leq \mathrm{Inst\ D}}$$

4.4.4 Typage des expressions JAVA

On supposera que FEnv(C,P) désigne l'environnement de types des champs définis dans la classe C du programme P. Ces champs peuvent être définis dans la classe C elle-même ou bien dans l'une de ses superclasses. FEnv(C,P) est un environnement semblable à ceux qui ont été utilisés pour le typage de C et qui seront utilisés à nouveau ici pour les variables locales (paramètres des méthodes ou variables déclarées dans un bloc). Toutefois, l'environnement FEnv(C,P) fournira deux informations : le type du champ et aussi son statut (champ d'instance ou champ statique). Ceci permettra de vérifier qu'un champ d'instance n'est pas utilisé dans une méthode statique. Par contre, on n'utilisera pas ici la protection. Tous les champs seront traités ici comme des champs publics.

De façon similaire, on désignera par MEnv(C,P) un environnement associant à chaque nom de méthode m utilisable dans la classe C, une liste de triplets comportant une **signature**, un type résultat et un statut pour cette méthode. La signature d'une méthode est la liste des types de ses arguments.

Le calcul de MEnv(C,P), qui dans un vérificateur de type pour JAVA sera fait une seule fois pour chaque classe, sera l'occasion de vérifier que les définitions de méthodes rencontrées satisfont bien les contraintes imposés par JAVA, c'est-à-dire par notamment :
– une méthode d'instance ne doit pas masquer une méthode statique ayant la même signature

– une méthode statique ne doit pas masquer une méthode d'instance ayant la même signature
– Lorsqu'une méthode en masque une autre, le type du résultat doit être le même dans les deux méthodes.

Le typage d'une expression sera exprimé par une formule

$$\text{P,C,E,s} \vdash \text{e : t}$$

où
– P est le programme considéré
– C est la classe considérée dans le programme P
– E est l'environnement donnant le types des variables locales susceptibles d'être utilisées dans l'expression e
– s est le statut de la méthode dans laquelle apparaît l'expression e. Cette information est nécessaire pour savoir si on a le droit d'utiliser this et super et quel sens on doit leur attribuer ainsi que pour vérifier la correction des accès aux champs.

$$\frac{}{\text{P,C,E,inst} \vdash \text{this : Inst C}}\ \text{(This)}$$

$$\frac{\text{P} \vdash \text{C extends D}}{\text{P,C,E,inst} \vdash \text{super : Inst D}}\ \text{(Super)}$$

$$\frac{\text{E(x)=t}}{\text{P,C,E,s} \vdash \text{x : t}}\ \text{(VarLocale)}$$

$$\frac{\text{E(x) non défini} \qquad \text{t,nonstatic=FEnv(P,C)(x)}}{\text{P,C,E,nonstatic} \vdash \text{x : t}}\ \text{(Champ1)}$$

$$\frac{\text{E(x) non défini} \qquad \text{t,static= FEnv(P,C)(x)}}{\text{P,C,E,s} \vdash \text{x : t}}\ \text{(Champ2)}$$

$$\frac{}{\text{P,C,E,s} \vdash \text{new D() : Inst D}}\ \text{(New)}$$

$$\frac{\text{P,C,E,s} \vdash \text{e : Inst C'} \qquad \text{t,\_=FEnv(P,C')(x)}}{\text{P,C,E,s} \vdash \text{e.x : t}}\ \text{(AccesChamp)}$$

$$\frac{\text{P,C,E,s} \vdash e_1 : t_1 \qquad \text{P,C,E,s} \vdash e_2 : t_2 \qquad \text{assignable}(e_1) \qquad \text{P} \vdash t_2 \leq t_1}{\text{P,C,E,s} \vdash e_1 = e_2 : t_1}\ \text{(Affectation)}$$

$$\frac{\text{P,C,E,s} \vdash \text{e : Inst D} \quad (t_1, ..., t_n),\text{t,\_=MEnv(P,D)(m)} \quad \text{P,C,E,s} \vdash e_i : t'_i \quad \forall i \;\; \text{P} \vdash t'_i \leq t_i}{\text{P,C,E,s} \vdash \text{e.m}(e_1, ..., e_n) \;:\; \text{t}} \text{ (AppelMethode1)}$$

$$\frac{(t_1, ..., t_n),\text{t,static=MEnv(P,D)(m)} \quad \text{P,C,E,s} \vdash e_i : t'_i \quad \forall i \;\; \text{P} \vdash t'_i \leq t_i}{\text{P,C,E,s} \vdash \text{D.m}(e_1, ..., e_n) \;:\; \text{t}} \text{ (AppelMethode2)}$$

4.4.5 Typage des instructions JAVA

$$\frac{\text{P,C,E,s} \vdash \text{decls : E'} \quad \text{P,C,E',s} \vdash \text{stats}}{\text{P,C,E,s} \vdash \{\text{decls stats }\}} \text{ (Bloc)}$$

$$\frac{\text{P,C,E,s} \vdash \text{e : t}}{\text{P,C,E,s} \vdash (\text{e};)} \text{ (Comput)}$$

$$\frac{\text{P,C,E,s} \vdash \text{e : bool} \quad \text{P,C,E,s} \vdash \text{stat}_1 \quad \text{P,C,E,s} \vdash \text{stat}_2}{\text{P,C,E,s} \vdash \text{if (e) stat}_1 \text{ else stat}_2} \text{ (If)}$$

$$\frac{\text{P,C,E,s} \vdash \text{e}_1 : \text{t}_1 \quad \text{P,C,E,s} \vdash \text{e}_2 : \text{boolean} \quad \text{P,C,E,s} \vdash \text{e}_3 : \text{t}_3 \quad \text{P,C,E,s} \vdash \text{stat}}{\text{P,C,E,s} \vdash \text{for } (\text{e}_1 ;\text{e}_2 ;\text{e}_3) \text{ stat}} \text{ (For)}$$

$$\frac{\text{E(return)} = \text{void}}{\text{P,C,E,s} \vdash \text{return}} \text{ (Return0)}$$

$$\frac{\text{E(return)} = \text{t} \quad \text{P,C,E,inst} \vdash \text{e : t'} \quad \text{P} \vdash \text{t'} \leq \text{t}}{\text{P,C,E,s} \vdash \text{return(e)}} \text{ (Return)}$$

$$\frac{}{\text{P,C,E,s} \vdash \text{Nop}} \text{ (Nop)}$$

$$\frac{\text{P,C,E,s} \vdash \text{stat} \quad \text{P,C,E,s} \vdash \text{stats}}{\text{P,C,E,s} \vdash \text{stat stats}} \text{ (Sequence)} \qquad \frac{}{\text{P,C,E,s} \vdash \text{/* vide */}} \text{ (Seq0)}$$

4.4.6 Typage des déclarations de variables locales

Le typage des déclarations de variables locales augmente l'environnement de type des variables locales.

$$\frac{}{\text{P,C,E,s} \vdash \text{t x : } [\text{x :t}] \oplus \text{E}} \text{ (SimpleDeclVar)}$$

$$\frac{\text{P,C,E,s} \vdash e : t_1 \qquad \text{P} \vdash t_1 \leq t}{\text{P,C,E,s} \vdash \text{t x = e : } [\text{x :t}] \oplus \text{E}} \text{ (InitDeclvar)}$$

$$\frac{\text{P,C,E,s} \vdash \text{decl : E'} \qquad \text{P,C,E',s} \vdash \text{decls : E''}}{\text{P,C,E',s} \vdash \text{decl decls : E''}} \text{ (SeqDecl)}$$

$$\frac{}{\text{P,C,E,s} \vdash \text{/*vide*/ : E}} \text{ (Decl0)}$$

4.4.7 Typage des déclarations de champs et de méthodes

Le typage des déclarations de champs obéit aux mêmes règles que celui des déclarations de variables locales à ceci près qu'elles n'enrichissent pas l'environnement local mais l'environnement des champs fournit par FEnv. On peut noter cependant que le résultat de FEnv ne dépend pas du typage des initialisations. On peut donc implémenter FEnv indépendamment de la vérification de types puis utiliser ensuite la fonction de typage des expressions pour vérifier que les initialisations sont bien correctes. Il en sera de même pour MEnv et les déclarations de méthodes.
Par conséquent, le typage des déclarations de méthodes se présente comme une simple vérification de correction et sera exprimé par des formules de la forme

$$\text{P,C} \vdash \text{mdecl}$$

Voici les règles définissant la correction des déclarations de méthodes :

$$\frac{\text{P,C,}[x_1 : t_1; ...x_n : t_n; return : t]\text{,inst} \vdash \text{bloc}}{\text{P,C} \vdash \text{t m}(t_1x_1, ..., t_nx_n) \text{ bloc}} \text{ (MethDecl1)}$$

$$\frac{\text{P,C,}[x_1 : t_1; ...x_n : t_n; return : t]\text{,static} \vdash \text{bloc}}{\text{P,C} \vdash \text{static t m}(t_1x_1, ..., t_nx_n) \text{ bloc}} \text{ (MethDecl2)}$$

4.4.8 Le problème de la surcharge

Levée de la surcharge

On appelle signature un n-uplet de types. Lorsqu'une méthode est déclarée sous la forme

$$t\ m(t_1\ x_1, ..., t_n\ x_n)\ bloc$$

sa **signature de définition** est $(t_1, ..., t_n)$.
Lorsque cette méthode est ensuite appelée dans une expression

$$e.m(e_1, ..., e_n)$$

ses arguments $(e_1, ..., e_n)$ ont des types $(u_1, ..., u_n)$ qui constituent une **signature d'appel.**
On dira qu'une signature $(u_1, ..., u_n)$ est compatible avec une signature $(t_1, ..., t_n)$ ou bien encore qu'elle est plus petite ou plus **spécifique** que celle-ci si pour tout i, u_i est un sous-type de t_i

Pour calculer le type d'un appel de méthode $e.m(e_1, ..., e_n)$ où e est un objet de classe C, on procède ainsi :

1. On calcule la signature d'appel $(u_1, ..., u_n)$.
2. On recense dans la hiérarchie de classes qui est au-dessus de C toutes les méthodes dont la signature de définition est plus grande que $(u_1, ..., u_n)$.
3. Parmi celles-ci, on regarde s'il y en a une dont la signature est plus petite que toute les autres. Si c'est le cas, on la choisit. Sinon, le programme est refusé.
4. Le type de retour de la méthode choisie donne le type de l'expression $e.m(e_1, ..., e_n)$.

On donne maintenant un programme CAML implémentant cet algorithme.

Fonction pour rechercher la définition d'une classe :

```
(* val getclassdef : string -> javaprog -> javaclassdecl *)
let getclassdef clname   (JavaProg classlist) =
 if clname="Object" then {classname="Object"; classfather= Nopt;
                    fielddecls=[]; methdecls=[];}  else
 List.find (function {classname=clname1} -> clname1=clname) classlist;;
```

Fonctions pour construire une représentation de la hiérarchie des classes :

```
(*  val ancetres : string -> javaprog -> string list   *)
(* On detecte les definitions de classes  circulaires *)
let ancetres classname prog =
    let rec anc cn cl =
       match (getclassdef  cn prog).classfather
       with Nopt   ->     "Object"::cl
         |  Some name  ->  if List.mem name cl
                             then raise (Failure ("circularite"))
                             else anc name (cn::cl)
in List.rev (anc classname []);;

(* val table_ancetres : javaprog -> (string * string list) list *)
let table_ancetres (JavaProg(classlist) as prog) =
  List.map
    (function {classname=clname} -> clname , ancetres clname prog)
    classlist;;
```

Fonctions pour calculer la relation de sous-typage (on se limite ici aux types qui correspodent à des classes en excluant les types de base comme int et float) :

```
(* val sousclasse : ('a * 'b list) list -> 'a -> 'b -> bool *)
let rec sousclasse table classname1 classname2 =
  List.mem classname2 (List.assoc classname1 table);;

(* val soustype :
  (javaident * javaident list) list ->  javatype -> javatype -> bool
  *)
let rec soustype table  ty1 ty2 =
(ty1=ty2) ||
match (ty1,ty2)
with
|     JavaIdType classname1 , JavaIdType classname2
          -> sousclasse table classname1 classname2
|     _,_   -> false
;;
```

Fonction pour calculer la relation entre signatures :

```
(* val signature_acceptable :
  (javaident * javaident list) list ->  javatype list -> javatype list -> bool
*)
let signature_acceptable table argtypes paramtypes =
try
  List.for_all (function (x,y) -> soustype table  x y)
              (List.combine argtypes paramtypes)
with Invalid_argument "List.combine" -> false;;

(* val signature : javamethdecl -> javatype list  *)
let signature methdef = List.map snd methdef.methparams;;
```

Fonctions pour les définitions de méthodes compatibles avec une signature d'appel donnée :

```
(* val methodes_compatibles :
  javaprog ->
  (javaident * javaident list) list ->
  string ->
  javatype list ->
  javaident ->
  (javaident * javamethdecl) list
*)
let methodes_compatibles prog table  methname  argtypes classname
=
  let hierarchie = classname::List.assoc classname table in
  let  methodes_compatibles_dans_classe  mname  argtypes  classname
     = let methlist =
          List.filter
             (function {methname=mn} as methdef
               -> mn=mname &&
                    signature_acceptable table argtypes (signature methdef))
```

```
            ((getclassdef classname prog).methdecls)
       in List.map (function meth -> classname, meth) methlist
   in
    List.flatten(List.map (methodes_compatibles_dans_classe methname argtypes)
                      hierarchie);;
```

Fonction pour trouver la définition de méthode plus spécifique que toutes les autres :

```
(* val trouver_methode :
  javaprog ->
  javaident ->
  string ->
  javatype list -> javaident * javamethdecl
*)
let trouver_methode prog classname methname argtypes =
  let table = table_ancetres prog in
  let methodes = methodes_compatibles prog table
                          methname  argtypes classname  in
  try
    List.find
    (function (clname,methdef) ->
        List.for_all
           (function (clname1,methdef1)
                -> signature_acceptable table
                        (JavaIdType clname::signature methdef)
                        (JavaIdType clname1::signature methdef1))
           methodes)
     methodes
 with Not_found -> raise (Failure ("La methode " ^ methname ^ " n'existe pas ou est
                               definie de facon ambigue"))
;;
```

Nouvelle formulation des règles d'appel

Nous supposerons que unique(m,P,C,$(t_1, ..., t_n)$)) retourne un couple formé des deux informations suivantes :

- le type de retour de l'unique méthode résultant du calcul de levée de la surcharge pour un appel de la méthode m avec des types d'arguments $(t_1, ..., t_n)$) dans la classe C du programme P.
- le statut de cette unique méthode

Lorsque la levée de la surcharge échoue, unique ne retournera pas de résultat.

$$\frac{\text{P,C,E,s} \vdash \text{e : Inst D} \qquad \forall i \;\; \text{P,C,E,s} \vdash e_i : t_i \qquad \text{t,\_=unique(m,P,D,}(t_1, ..., t_n)\text{)}}{\text{P,C,E,s} \vdash \text{e.m}(e_1, ..., e_n) \; : \; \text{t}} \text{ (AppelMethode1)}$$

$$\frac{\text{P,C,E,s} \vdash e_i : t_i \qquad \text{t,static} = \text{unique(m,P,D,}(t_1, ..., t_n)\text{)}}{\text{P,C,E,s} \vdash \text{D.m}(e_1, ..., e_n) \ : \ \text{t}} \ \text{(AppelMethode2)}$$

Chapitre 5

Évaluation

5.1 Évaluation à l'aide d'environnements

Un environnement d'évaluation est une table qui permet d'associer aux variables d'un programme les valeurs qui leur correspondent. En CAML, les variables sont introduites par la construction `let` (éventuellement `let rec`) et par la construction fonctionnelle (`function`) et ses variantes comme par exemple le `match with`. On mémorisera donc la valeur d'une variable dans un environnement essentiellement dans deux circonstances.

- Quand on évaluera une expression de la forme
  `let x = e1 in e2`
  Dans ce cas, on évaluera d'abord `e1` en une valeur `v` puis on notera dans l'environnement que `x` a la valeur `v` pour évaluer `e2`.
- Quand on évaluera un appel de fonction `f(e1)` où `f = function x -> e`.
  Dans ce cas, on évaluera d'abord `e1`, ce qui produira une valeur `v`, puis on notera dans l'environnement que `x` a la valeur `v` pour évaluer `e`.

On remarque qu'avec cette façon de procéder, l'expression `let x = e1 in e2` est équivalente à `(fun x -> e2) e1`.

5.1.1 Règles d'évaluation pour CAML

Les règles que nous allons donner permettent de définir la valeur des expressions CAML. Comme nous nous limitons ici à un sous-langage qui ne comporte pas de définition de nouveaux types, les valeurs vont être principalement des valeurs de base (entiers, booléens) ou des paires. Cependant, il reste un type d'expressions pour lequel il n'est pas évident de définir une valeur : il s'agit des expressions fonctionnelles. Le point de vue que nous adopterons ici est que la valeur d'une fonction consiste en son texte et son environnement de définition. Le texte d'une fonction permet de savoir ce qu'il faut faire pour appliquer la fonction à un argument. Toutefois, ce texte n'est pas suffisant car le corps de la fonction peut contenir des variables libres dont la définition doit être cherchée dans l'environnement qui existe au momeny où la fonction est définie. Il faut donc adjoindre cet environnement au texte de la fonction pour pouvoir appliquer correctement celle-ci. Une paire formée d'un environnement et d'un texte sera appelée une **fermeture**.

$$\frac{\text{E(x)=v}}{\text{E} \vdash \text{x} \Rightarrow \text{v}} \text{ (Var)}$$

$$\frac{\text{E} \vdash e_1 \Rightarrow v_1 \quad \text{E} \vdash e_2 \Rightarrow v_2}{\text{E} \vdash (e_1, e_2) \Rightarrow (v_1, v_2)} \text{ (Paire)}$$

$$\frac{\text{E} \vdash e_1 \Rightarrow \text{bool} \qquad \text{E} \vdash e_2 \Rightarrow \text{v}}{\text{E} \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow \text{v}} \text{ (Cond1)}$$

$$\frac{\text{E} \vdash e_1 \Rightarrow \text{false} \qquad \text{E} \vdash e_3 \Rightarrow \text{v}}{\text{E} \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow \text{v}} \text{ (Cond2)}$$

$$\frac{}{\text{E} \vdash (\text{fun x} \rightarrow \text{e}) \Rightarrow < \text{E}, (\text{fun x} \rightarrow \text{e}) >} \text{ (Fun)}$$

$$\frac{\text{E} \vdash e_1 \Rightarrow < \text{E', fun x} \rightarrow \text{e} > \qquad \text{E} \vdash e_2 \Rightarrow v_2 \qquad [x, v_2] \oplus \text{E'} \vdash \text{e} \Rightarrow \text{v}}{\text{E} \vdash (e_1\ e_2) \Rightarrow \text{v}} \text{ (App)}$$

$$\frac{\text{E} \vdash e_1 \Rightarrow v_1 \qquad [x, v_1] \oplus \text{E} \vdash e_2 \Rightarrow v}{\text{E} \vdash \text{let x} = e_1 \text{ in } e_2 \Rightarrow \text{v}} \text{ (Let)}$$

La prise en compte de la récursivité introduit un problème supplémentaire. Comme une fonction récursive doit se connaitre elle-même, la fermeture qui la représente doit contenir la définition de la fonction elle-même. On est donc amené à construire des environnement “en boucle”.

$$\frac{\text{E'}= [f, < E', (\text{funx} \rightarrow e) >] \oplus \text{E} \ \vdash e_2 \Rightarrow v}{\text{E} \vdash \text{let rec f x} = e_1 \text{ in } e_2 \Rightarrow \text{v}} \text{ (Letrec)}$$

5.1.2 Un évaluateur de CAML en CAML

```
open Camlsyntax;;
open Environment;;

type caml_val = Intval of int
              | Boolval of bool
```

```
              | Pairval of caml_val * caml_val
              | Clo of (string , caml_val) environment * exp;;


let unop op e =
match (op,e)
with     "fst" , Pairval (x,y) -> x
  |      "snd" , Pairval (x,y) -> y
  |      "not" , Boolval b -> Boolval (not b)
  |   _  -> failwith "unop: wrong type";;

let binop op e1 e2=
match (op,e1,e2)
with
  "+", Intval m , Intval n -> Intval (m+n)
| "-", Intval m , Intval n -> Intval (m-n)
| "*", Intval m , Intval n -> Intval (m*n)
| "/", Intval m , Intval n -> Intval (m/n)
| "=", Intval m , Intval n -> Boolval (m=n)
| "<", Intval m , Intval n -> Boolval (m<n)
| ">", Intval m , Intval n -> Boolval (m>n)
| "<=", Intval m , Intval n -> Boolval (m<=n)
| ">=", Intval m , Intval n -> Boolval (m>=n)
|   _  -> failwith "binop: wrong types";;

let rec eval env e =
match e
with
  Int n -> Intval n
| Bool b -> Boolval b
| Pair (e1,e2) -> Pairval(eval env e1, eval env e2)
| Var x -> (try get x env
            with Not_found -> failwith "unbound variable")
| If (c,e1,e2) -> (match eval env c with
                          (Boolval true) -> eval env e1
                        | (Boolval false) -> eval env e2
                        | _ -> failwith "wrong types")
| Fun (x,e) as f -> Clo(env,f)
| App(App(Op op,e1),e2) -> binop op (eval env e1) (eval env e2)
| App(Op op,e) -> unop op (eval env e)
| App (e1,e2) -> (match (eval env e1, eval env e2) with
                         (Clo(env',Fun (x,e)),v)
                         -> eval (push (x,v) env') e
                       | _ -> failwith "wrong types")
| Let (x,e1,e2) -> let v = eval env e1
                         in eval (push (x,v) env) e2
| Letrec (f,e1,e2)
    -> (match e1 with
        Fun _ -> let rec env' = (f,Clo(env',e1))::env
```

```
                in eval env' e2
      | _ -> failwith "illegal recursive definition")
;;
```

5.2 Évaluation des traits impératifs

L'évaluation de traits impératifs fait nécessairement appel à une mémoire susceptible d'être modifiée par des opérations telles que l'affectation. Nous représenterons cette mémoire de façon abstraite à travers un type que nous appellerons **store**. Un store peut être implémenté de différentes façons mais nous ferons abstraction pour le moment de toute implémentation. Nous verrons un store comme un ensemble de **locations** (représentées par un type **loc** qu'on peut interpréter comme des adresses dans une mémoire). Parmi les adresses, certaines contiennent des valeurs (représentées par un type **val**, , d'autres sont inoccupées. La valeur contenue à une adresse est soit une valeur immédiate (entiers, flottants, etc.) soit un pointeur c'est-à-dire une autre adresse.
Nous utiliserons trois opérations :

- **get** : store → loc → val
  Cette opération permet d'obtenir la valeur se trouvant dans une adresse.
- **new** : store → store * loc
  Cette opération permet d'obtenir une adresse non encore utilisée. Elle renvoie un nouveau store et la nouvelle adresse.
- **set** : store → loc → val → store
  Cette opération permet de mettre une valeur à une adresse. Elle renvoie un nouveau store.

Souvent, on combinera **new** et **set** dans une opération

- **set_new** : store → val → store
  set_new s v = let s',i = new s in set s' i

L'évaluation d'un morceau de programme aura aussi besoin d'un environnement. Comme précédemment, un environnement est une table associant identificateurs et valeurs . Les environnements admettent essentiellement les opérations get et push. On y ajoutera pushl qui permet d'ajouter en une seule fois un ensemble de valeurs à un environnement..

Contrairement à ce qui se passe pour les langages fonctionnels dont les environnement associent directement des valeurs aux identificateurs, pour les langages impératifs, cette association se fait en deux temps. L'environnement associe une adresse à un identificateur et le store permet ensuite de connaître la valeur contenue à cette adresse.

Notons enfin que, contrairement à ce qui se passe en CAML, les fonctions de C sont toujours définies statiquement et non dynamiquement. Il n'est donc pas nécessaire d'utiliser ici des fermetures. Une fonction sera donc représentée simplement par son corps et la liste de ses paramètres.

5.2.1 Règles d'évaluation pour C

Dans les règles d'évaluation qui suivent, nous noterons simplement E(s) et S(i) pour "get E s" et "get S i". Nous noterons également $S\ [i \leftarrow v]$ pour "set S i v" et $S\ [new \leftarrow v]$ pour "set_new S v".

Les jugements d'évaluation auront des formes différentes suivant que la construction qui est évaluée est une déclaration, une instruction ou une expression.
Une déclaration évaluée dans un environnement E et un store S, produit un nouvel environnement et un nouveau store, ce qui s'écrira

$$\text{E,S} \vdash \text{decl} \Rightarrow \text{E',S'}$$

Une instruction évaluée dans un environnement E et un store S, produit uniquement un nouveau store (l'environnement étant inchangé), ce qui s'écrira

$$\text{E,S} \vdash \text{stat} \Rightarrow \text{S'}$$

Enfin, une expression évaluée dans un environnement E et un store S, produit uniquement un nouveau store et une adresse (l'adresse où est rangée la valeur de l'expression) ce qui s'écrira

$$\text{E,S} \vdash \text{exp} \Rightarrow \text{S',i}$$

Evaluation des déclarations

$$\frac{\text{S',i} = \text{new S}}{\text{E,S} \vdash \text{t x} \Rightarrow [x, i] \oplus \text{E} \ , \ \text{S'}} \ \text{(VARDECL)}$$

$$\frac{\text{E'} = [f, Fun((x_1, ..., x_n), body)] \oplus \text{E}}{\text{E,S} \vdash t\ f(t_1\ x_1, ..., t_n\ x_n)\{body\} \Rightarrow \text{E'} \ , \ \text{S}} \ \text{(FUNDECL)}$$

Evaluation des instructions

$$\frac{\text{E,S} \vdash \text{decls} \Rightarrow \text{E'} \ , \ \text{S'} \qquad \text{E',S'} \vdash \text{stats} \Rightarrow \text{S"}}{\text{E,S} \vdash \{\text{decls ;stats}\} \Rightarrow \text{S"}} \ \text{(BLOCK)}$$

$$\frac{\text{E,S} \vdash \text{e} \ \Rightarrow \text{S'} \ , \ \text{i} \qquad \text{S'(i)}\neq 0 \qquad \text{E,S'} \vdash s_1 \Rightarrow \text{S"}}{\text{E,S} \vdash \text{if e then } s_1 \text{ else } s_2 \Rightarrow \text{S"}} \ \text{(IF1)}$$

$$\frac{\text{E,S} \vdash \text{e} \ \Rightarrow \text{S'} \ , \ \text{i} \qquad \text{S'(i)}=0 \qquad \text{E,S'} \vdash s_2 \Rightarrow \text{S"}}{\text{E,S} \vdash \text{if e then } s_1 \text{ else } s_2 \Rightarrow \text{S"}} \ \text{(IF2)}$$

$$\frac{\text{E,S} \vdash \text{e} \ \Rightarrow \text{S}_1 \ , \ \text{i} \qquad \text{S}_1\text{(i)}\neq 0 \qquad \text{E,S}_1 \vdash \text{s} \ \Rightarrow \text{S}_2 \qquad \text{E,S}_2 \vdash \text{while (e) s} \Rightarrow \text{S}_3}{\text{E,S} \vdash \text{while (e) s} \Rightarrow \text{S}_3} \ \text{(WHILE1)}$$

$$\frac{\text{E,S} \vdash \text{e} \Rightarrow \text{S}_1 \ , \ \text{i} \qquad \text{S}_1\text{(i)}=0}{\text{E,S} \vdash \text{while (e) s} \ \Rightarrow \text{S}_1} \ \text{(WHILE2)}$$

Pour les boucles for, nous distinguerons le cas où il n'y a pas d'initialisation et le cas où il y en a une.

$$\frac{\begin{array}{c}\text{E,S} \vdash e_2 \Rightarrow \text{S}_1\text{,i} \qquad S_1(i) \neq 0 \qquad \text{E,S}_1 \vdash s \Rightarrow \text{S}_2 \\ \text{E,S}_2 \vdash e_3 \Rightarrow \text{S}_3 \qquad \text{E,S}_3 \vdash \text{for } (\,;e_2\,;e_3)\ \text{s} \Rightarrow \text{S}_4\end{array}}{\text{E,S} \vdash \text{for } (\,;e_2\,;e_3)\ \text{s} \Rightarrow \text{S}_4}\ \text{(FOR1)}$$

$$\frac{\text{E,S} \vdash e_1 \Rightarrow \text{S'i} \qquad \text{E,S'} \vdash \text{for } (\,;e_2\,;e_3)\ \text{s} \Rightarrow \text{S"}}{\text{E,S} \vdash \text{for } (e_1\,;e_2\,;e_3)\ \text{s} \Rightarrow \text{S"}}\ \text{(FOR2)}$$

$$\frac{\text{E,S} \vdash \text{e} \Rightarrow S'\ \text{, i}}{\text{E,S} \vdash \text{Return(e)} \Rightarrow S'\ \ [E(return)\ \leftarrow\ S'(i)]}\ \text{(RETURN)}$$

$$\frac{}{\text{E,S} \vdash \text{Nop} \Rightarrow \text{S}}\ \text{(NOP)}$$

Il faut aussi une règle pour les suites d'instructions.

$$\frac{\text{E,S} \vdash \text{stat} \Rightarrow \text{S'} \qquad \text{E,S'} \vdash \text{stats} \Rightarrow \text{S"}}{\text{E,S} \vdash \text{stat stats} \Rightarrow \text{S"}}\ \text{(STATS)}$$

Evaluation des expressions

$$\frac{\text{E(x)=i}}{\text{E,S} \vdash \text{x} \Rightarrow \text{S,i}}\ \text{(VAR)}$$

$$\frac{\text{E,S} \vdash \text{e} \Rightarrow \text{S'i} \qquad \text{S'(i)=j}}{\text{E,S} \vdash \text{*e} \Rightarrow \text{S'j}}\ \text{(INDIR)}$$

$$\frac{\text{E,S} \vdash e_1 \Rightarrow \text{S'i} \qquad \text{E,S'} \vdash e_2 \Rightarrow \text{S"j}}{\text{E,S} \vdash e_1 := e_2 \Rightarrow S''\ \ [i\ \leftarrow\ S''(j)]\text{j}}\ \text{(ASSIGN)}$$

La règle définissant les appels de fonctions est une règle cruciale pour l'évaluation et c'est de loin la plus complexe. Nous en donnerons d'abord une version "par référence" qui n'est pas celle de C puis nous donnerons la règle "par valeur", qui est celle de C.

$$\frac{\begin{array}{c}\text{E(f)} = \text{Fun}((x_1, ..., x_n)\text{,body}) \\ \text{S'i} = \text{new S} \qquad \text{E,S'} \vdash e_1 \Rightarrow S_1\text{,}i_1 \ \ldots\ \text{E,}S_{n-1} \vdash e_n \Rightarrow S_n\text{,}i_n \\ [(x_1, i_1), ..., (x_n, i_n), (return, i)] \oplus \text{E}\ \ \text{,S}_n \vdash \text{body} \Rightarrow \text{S"}\end{array}}{\text{E,S} \vdash \text{f}(e_1, ..., e_n) \Rightarrow \text{S"i}}\ \text{(CALL\_REF)}$$

$$\frac{\begin{array}{c}\text{E(f)} = \text{Fun}((x_1,...,x_n),\text{body}) \\ \text{S',i} = \text{new S} \quad \text{E},S' \vdash e_1 \Rightarrow S_1,i_1 \ ... \ \text{E},S_{n-1} \vdash e_n \Rightarrow S_n,i_n \\ (j_1,...,j_n),\ S'' = \text{news } S_n \quad S'''=S''\ [j_k \leftarrow S''(i_k)] \\ [(x_1,j_1),...,(x_n,j_n),(return,i)] \oplus \text{E}\ \ ,S''' \vdash \text{body} \Rightarrow S''''\end{array}}{\text{E},S \vdash \text{f}(e_1,...,e_n) \Rightarrow S'''',\text{i}} \text{ (CALL\_VAL)}$$

5.2.2 Un évaluateur pour C

La représentation du store peut être envisagée de différentes façons. On peut représenter les stores comme des fonctions qui, à une adresse, associent une valeur. C'est la représentation la plus abstraite. On peut aussi les représenter comme des tableaux, ce qui est plus proche d'une véritable implantation. De façon à laiser la porte ouverte à ces différentes représentations, nous allons définir un type de module `Store` qui définira la fonctionnalité des stores tout en laissant ouvert le choix de leur représentation.

```
module type Store =
  sig
   type 'a store
   type loc
   exception Store_exc of string
   val get : 'a store -> loc -> 'a
   val set : 'a store -> loc -> 'a -> 'a store
   val news : 'a store -> 'a store * loc
   val set_new : 'a store -> 'a -> 'a store * loc
   val initial_store : loc -> 'a -> 'a store
  end
```

Ce module définit un type `'a store` permettant de mémoriser des valeurs de type quelconque et un type `loc` correspondant aux adresses. La représentation des types `'a store` et `loc` n'est pas précisée. Il s'agit de types abstraits.
Les éléments du store seront accessibles à travers leur adresse représentée par un entier. On se donne donc une fonction `get` permettant d'accéder à la valeur qui se trouve à une adresse donnée. De même, on se donne une fonction `set` qui permet de stocker une valeur à une certaine adresse.
Par ailleurs, on suppose qu'un store garde la trace de toutes les adresses occupées et est capable de fournir sur demande une adresse inoccupée. C'est le rôle de la fonction `news` qui retourne le nouveau store ainsi obtenu et cette nouvelle adresse. La fonction `set_new` combine `news` et `set`.
La fonction `get` peut échouer si on lui passe une adresse inexistante. De même, la fonction `set_new` peut échouer s'il n'y a plus d'adresse disponible. Dans ces deux cas, l'exception `Store_exc` sera déclanchée. Enfin, la fonction `initial_store` permet de créer un nouveau store d'une taille donnée dont toutes les mémoires seront initialisée par une valeur donnée.
On donne ci-dessous deux implantations possibles de ce modules. La première utilise une représentation fonctionnelle :

```
module FunStore : Store =
 struct
 type 'a store = int -> 'a
 type loc = int
```

```
 exception Store_exc of string
 let get store i = store i
 let new_int =
  let c = ref(-1) in fun () -> c:=!c+1;!c
 let set store i a = (fun j -> if j=i then a else store j)
 let news store = let i= new_int() in store,i
 let set_new store a = let i=new_int()
                       in (fun j -> if j=i then a else store j),i
 let initial_store store_size initial_value =
   fun i -> raise (Store_exc ("Undefined location: " ^ string_of_int i))
 end
```

La seconde utilise des tableaux :

```
module Arraystore : Store =
 struct
 type 'a store = {set: int -> 'a -> 'a store;
                  news: unit -> 'a store * int;
                  set_new: 'a -> 'a store * int;
                  get: int  -> 'a}
 type loc = int
 exception Store_exc of string
 let set store i a = store.set i a
 let news store = store.news()
 let set_new store a = store.set_new a
 let get store i = store.get i
 let undef i = Store_exc ("Undefined location: " ^ string_of_int i)
 let overflow = Store_exc "Store Overflow"
 let initial_store  store_size initial_value =
  let rec mem =
   let m =  Array.create store_size initial_value
   and new_loc = let c = ref(-1)
                 in fun () -> if !c < store_size-1
                                  then c:=!c+1;!c
                                  else raise overflow
   in {set= (fun i a -> try m.(i)<-a;mem
                        with _ ->  raise (undef i)
       news= (fun () -> let i = new_loc() in (mem,i));
       set_new = (fun a -> let i = new_loc()
                           in m.(i)<- a;(mem,i));
       get= (fun i -> m.(i))}
  in mem
 end
```

Par souci d'homogénéité, on procédera de la même façon pour les environnements.

```
module type Env  =
  sig
   type ('a , 'b) environment
   val push : ('a * 'b) -> ('a , 'b) environment  -> ('a , 'b) environment
   val pushl : ('a * 'b) list  -> ('a , 'b) environment  -> ('a , 'b) environment
   val get : 'a -> ('a , 'b) environment -> 'b
```

```
   val get_list : ('a , 'b) environment -> ('a * 'b) list
   val initial_env : ('a , 'b) environment
  end
```

L'implementation la plus simple pourra être :

```
module Listenv  =
  struct
   type ('a , 'b) environment = ('a * 'b) list
   let push p env = p::env
   let pushl pl env = pl@env
   let get a env = List.assoc a env
   let get_list env = env
   let initial_env = []
  end
```

Nous pouvons définir à présent un module `Evaluator` qui fournira en particulier une fonction d'evaluation. Le type de ce module sera :

```
module type Evaluator =
  sig
   type memory_content =
     I of int | F of float | B of bool | V of unit | A of int
   type env_val = Adr of int
              | FunDef of string list * cstat
   exception Run_Error of string
   val eval_prog : cprog -> memory_content
  end
```

On y définit d'abord un type pour les valeurs contenues dans la mémoire. Ce sont soit des valeurs immédiates soit des adresses représentées par des entiers.
On y définit ensuite un type pour les valeurs stockées dans l'environnement qui sont soit des définitions de fonctions (stockées directement dans l'environnement) soit des adresses.
Enfin, on y définit une exception correspondant aux erreurs d'exécution et une fonction d'évaluation.

L'implantation de ce module sera paramétrée par les implantations de l'environnement et du store :

```
module Eval (E:Env) (S:Store) : Evaluator =
  struct
  ...
  end
```

```
module Eval (E:Env) (S:Store) : Evaluator =
  struct

type memory_content =
 I of int | F of float | B of bool | V of unit | A of int;;

type env_val = Adr of int
             | FunDef of string list * cstat;;
```

```
exception Run_Error of string;;


let eval_decl env store decl =
  match decl
  with VarDecl {varname=v; vartype= Voidtype}
    -> let store,i = S.set_new store (V())
       in  E.push (v,Adr i) env, store
  |  VarDecl {varname=v; vartype= Inttype}
    -> let store,i = S.set_new store (I 0)
       in  E.push (v,Adr i) env, store
  |  VarDecl {varname=v; vartype= Floattype}
    -> let store,i = S.set_new store (F 0.)
       in  E.push (v,Adr i) env, store
  |  VarDecl {varname=v; vartype= Pointertype(_)}
    -> let store,i = S.set_new store (A 0)
       in  E.push (v,Adr i) env, store

  | FunDecl d
    -> let params = map fst d.funparamstypes
       in E.push  (d.funname, FunDef(params ,d.funbody)) env , store
  |  _ -> env,store;;



let rec eval_decls env store decls =
  match decls
  with    [] -> env , store
  |   d::ds ->  let env',store' =  eval_decl env store d
                in eval_decls env' store' ds;;

let eval_unop =
function UMOINS -> (function (I n) -> I(n-1)
                           | _ -> raise (Run_Error("Uminus: wrong arg")))
  |      NOT  -> (function (I n) -> I(if n=0 then 1 else 0)
                           | _ -> raise (Run_Error("Not: wrong arg")));;


let eval_binop =
function PLUS -> (function (I m,I n) -> I(m+n)
                           | _ -> raise (Run_Error("Plus: wrong arg")))
  |      MOINS -> (function (I m,I n) -> I(m-n)
                           | _ -> raise (Run_Error("Plus: wrong arg")))

  |      MULT -> (function (I m,I n) -> I(m*n)
                           | _  -> raise (Run_Error("Plus: wrong arg")))

 |       DIV -> (function (I m,I n) -> I(m/n)
```

```
                        | _ -> raise (Run_Error("Plus: wrong arg")))

 |      EGAL -> (function (I m,I n) -> I(if m=n then 1 else 0)
                        | _ -> raise (Run_Error("Plus: wrong arg")))

 |      LT -> (function (I m,I n) -> I(if m<n then 1 else 0)
                        | _ -> raise (Run_Error("Plus: wrong arg")))

 |      GT -> (function (I m,I n) -> I(if m>n then 1 else 0)
                        | _ -> raise (Run_Error("Plus: wrong arg")))

 |      LE -> (function (I m,I n) -> I(if m<=n then 1 else 0)
                        | _ -> raise (Run_Error("Plus: wrong arg")))

 |      GE -> (function (I m,I n) -> I(if m>=n then 1 else 0)
                        | _ -> raise (Run_Error("Plus: wrong arg")));;



let rec eval_stat env store stat =
  match stat
  with Block(decls,stats)
   -> let env' , store' = eval_decls env store decls
      in eval_stats env' store' stats
  |  Comput e -> fst(eval_exp env store e)
  |  If(e,stat1,stat2)
    -> let store',i = eval_exp env store  e
       in (match S.get store' i
           with I 0 -> eval_stat env store' stat2
            |  I _ -> eval_stat env store' stat1
            |  _  -> raise (Run_Error ("If: wrong test")))
  |  For (Nothing,e2,e3,stat) as for_s
    -> let store2, i = eval_exp env store e2
       in (match S.get store2 i
           with I 0 -> store2
             |  I _ -> let store3 = eval_stat env store2 stat in
                       let store4,_ = eval_exp env store3 e3
                         in eval_stat  env store4  for_s
             | _ ->  raise (Run_Error ("For: wrong test")))

  |  For (e1,e2,e3,stat)
    -> let store',i = eval_exp env store  e1
       in eval_stat env store' (For(Nothing,e2,e3,stat))
```

```
  |  Return e
    -> let store',i = eval_exp env store e
       in (match  E.get "return" env
           with   Adr k -> S.set store' k (S.get store' i)
             |    _ ->  raise (Run_Error ("Bad return value")))
  |  Nop -> store

and eval_stats env store stats =
  match stats
  with    []   ->   store
   |    [s]    ->   eval_stat env store s
   |  Return e::stat'
      -> eval_stat env store (Return e)
   |  stat::stats'
      -> let store' = eval_stat env store stat
         in eval_stats env store' stats'



and eval_exp env store e =
 match e
 with Constexp (Int n)  -> S.set_new store (I n)
 |    Nothing -> S.set_new store (V())
 |    Varexp s
       -> (match E.get s env
           with Adr i -> store , i
              |  _ -> raise (Run_Error ("Variable " ^s^ " undefined")))
 |   Unopexp(op,e')
       -> let store' , i =  eval_exp env store e'
          in  S.set_new store'(eval_unop op  (S.get store' i))
 |   Binopexp(op,e1,e2)
       -> let store1,i = eval_exp env store e1 in
          let store2,j = eval_exp env store1 e2
          in  S.set_new store2 (eval_binop op (S.get store2 i,S.get store2 j))
 |   Index(e1,e2) ->  raise (Run_Error ("Arrays not implemented yet"))
 |   Indirection e
       -> let store1,i = eval_exp env store e in
          (match S.get store1 i
           with A(n) -> store1,n
           |  _ -> raise (Run_Error ("Wrong indirection")))
 |  Assign(e1,e2)
       -> let store1,i = eval_exp env store e1 in
          let store2,j = eval_exp env store1 e2
          in S.set store2 i (S.get store2 j) , i
(* VERSION PAR REFERENCE
 |  Call(f,args)
       -> let store1,i = S.news store in
          let store2 , args_adrs =  compute_args env store1 args in
          let args_adrs' = map (fun x -> Adr x) args_adrs in
```

```
          (match E.get f env
           with  FunDef(params,body)
            -> let store3 = eval_stat
                   (E.pushl (combine params args_adrs')
                   (E.push ("return",Adr i) env))
                   store2 body
               in store3,i
          | _ ->  raise (Run_Error (f ^ " is not a function")))
*)
(* VERSION PAR VALEUR *)
 |  Call(f,args)
       -> let store1,i = S.news store in
          let store2 , args_adrs =  compute_args env store1 args in
          let store3, locals_adrs = compute_locals env store2 args_adrs in
          let args_adrs' = map (fun x -> Adr x) locals_adrs in
          (match E.get f env
           with  FunDef(params,body)
            -> let store3 = eval_stat
                   (E.pushl (combine params args_adrs')
                   (E.push ("return",Adr i) env))
                   store3 body
               in store3,i
          | _ ->  raise (Run_Error (f ^ " is not a function")))


and  compute_args env store =
function []  -> store , []
  |  [e]  ->  let store1, i1 = eval_exp env store e
              in store1 , [i1]
  |  e::es  -> let store1, i1 = eval_exp env store e in
               let store2 ,is =  compute_args env store1 es
               in store2, i1::is

and compute_locals env store =
function []  -> store , []
  |  [i]  ->  let store1, j = S.set_new store (S.get store i)
              in store1, [j]
  |  i::is  -> let store1, j1 = S.set_new store (S.get store i) in
               let store2 ,js =  compute_locals env store1 is
               in store2, j1::js

;;

let store_size = ref 1000;;
let set_store_size n = store_size:=n;;

let eval_prog  (Prog decls) =
  let env', store' = eval_decls E.initial_env
                                (S.initial_store !store_size (A 0))
```

```
                                    decls
  in let m,i = eval_exp env' store' (Call("main",[]))
     in S.get m i;;
  end
```

On peut ensuite utiliser le module paramétré `Eval` en l'appliquant à un module de type `Env` et à un module de type `Store` :

```
# module M = Eval (Listenv) (Arraystore);;
module M :
  sig
    type memory_content =
      Evaluateur.Eval(Listenv)(Arraystore).memory_content =
      | I of int
      | F of float
      | B of bool
      | V of unit
      | A of int
    and env_val =
      Evaluateur.Eval(Listenv)(Arraystore).env_val =
      | Adr of int
      | FunDef of string list * Csyntax.cstat
    exception Run_Error of string
    val eval_prog : Csyntax.cprog -> memory_content
  end
```

Puis on peut utiliser le module `M` de la façon suivante :

```
M.eval_prog (parse_file "toto.c");;
```

où le fichier `toto.c` contient un programme C à évaluer.

Bibliographie

[1] H. Abelson and G.J. Sussman. *Structure et interpretation des programmes informatiques*. Interéditions, 1997.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilateurs : Principes, techniques et outils*. Dunod, 2000.

[3] K. Arnold and J. Gosling. *The Java language specification*. Addison Wesley, 1997.

[4] E. Chailloux, P. Manoury, and B. Pagano. *Objective Caml*. O'Reilly, 2000.

[5] J. Chazarain. *Programmer avec Scheme*. Internatinal Thomson Publishing, 1996.

[6] G. Cousineau and M. Mauny. *Approche fonctionnelle de la programmation*. Ediscience, 1995.

[7] B. Eckel. *Thinking in C++*. Disponible sur l'url : http ://www.bruceeckel.com/ThinkinhInCPP2e.html, 2000.

[8] B. Eckel. *Thinking in Java*. Disponible sur l'url : http ://www.eckelobjects.com/javabook.html, 2000.

[9] J. Gosling, B. Joy, and G. Steele. *The Java programming language*. Addison Wesley, 1997.

[10] S.P. Harbison and G. Steele. *Langage C : manuel de référence*. Masson, 1989.

[11] T. Hardin and V. Donzeau-Gouge Viguié. *Concepts et outils de programmation*. InterEditions, 1992.

[12] K. Jensen and N. With. *C : a referece manual*. Springer Verlag, 1974.

[13] X. Leroy and P. Weis. *Manuel de référence du langage Caml*. InterEditions, 1993.

[14] S. Prata. *C++ Primer Plus*. SAMS Publishing, 1998.

[15] C. Queinnec. *Les langages Lisp*. InterEditions, 1994.

[16] P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993.