Cours de Compilation-Exercices

Génération de code

Master d'Informatique M1 2009–2010

29 octobre 2009

1 Passage de paramètres

Décrire le comportement des programmes suivants lorsque les paramètres des procédures sont passés par valeur, par référence, par copie/restauration, ou par nom.

1. ```
   program param;
   var i : integer; a:array [1..2] of integer;
   procedure p ([MODE] x,y:integer);
       begin x:=x+y; i:=i+1; y:=y+1 end
   begin
   i:=1; a[1]:=1; a[2]:=1; p(a[i],a[i]); writeln(i,a[1],a[2]);
   end
   ```
2. ```
   program param;
       var i : integer; a:array [0..2] of integer;
       procedure swap ([MODE] x,y:integer);
       begin x:= x+a[i]; y:= x-a[i]; x:=x-y end
   begin i:=0; a[0]:=1 ; a[1]:=2 ; a[2]:=3;
   swap (i,a[i]); writeln(i,a[0],a[1],a[2]);
   swap (a[i],a[i]); writeln(a[0],a[1],a[2]);
   end
   ```
3. ```
   program part; var limit:integer;
   function summation ([MODE] lim:integer); var s : integer;
     begin s:=0; while lim>0 do begin s:=s+limit; limit:=limit-2 end
           summation:=s
     end;
   begin limit:=6; writeln ('Sum=',summation(limit)) end.
   ```

2 Passage de paramètres

(Examen janvier 2001)

1. Soit le programme:

   ```
   program param;
   var x : int;
   procedure proc(y:int);
      begin x:=x+y; y:=y+1 end;
   begin x:=2; proc(x); print(x) end;
   ```

   Quel est le résultat de ce programme lorsque le paramètre `y` de `proc` est passé :
   ```

(a) par valeur,

(b) par référence,

(c) par copie/restauration,

(d) par nom.

2. Quels sont les avantages et les inconvénients de ces différentes modes de passage de paramètres ?

3 Passage de paramètres en Java

Soit le programme Java :

```
class A {
    int x;
    A (int v) { x=v;}
    static void m (A a) {a = new A(2);}
    public static void main (String argv[]) {
        A b = new A(0); m (b);
        System.out.println(b.x); }
}
```

1. Quelle est le résultat de l'évaluation de ce programme ?
2. Justifier votre réponse en donnant les étapes de l'évaluation du programme (indiquer les variables allouées dans la pile et leur valeur en terme d'adresses des objets alloués dans le tas).
3. On considère maintenant le programme :

```
class B {
    A a;
    B (int v) { a=new A(0); }
    static void m (B b) {b.a = new A(2);}
    public static void main (String argv[]) {
        B c = new B(0); m (c);
        System.out.println(c.a.x); }
}
```

Donner le résultat de ce programme et expliquer les étapes de son évaluation.

4. Quel est le mode de passage de paramètres de Java ?

4 Procédures avec paramètres passés par valeur ou référence

Soit le programme:

```
program test;
var i,sum,facteur : integer;
procedure q (x:integer, y:integer);
  begin y:=facteur * x + y end;
begin
 i:=1; sum:=0; facteur:=2;
 while i <= 10 do begin q(i+1,sum); i:=i+1 end;
 writeln sum;
end
```

1. Indiquer comment seront allouées les variables globales et locales dans la pile.

2. Donner une séquence d'instructions pour le code compilé de ce programme.
3. Lorsqu'un paramètre est passé par référence, il faut pouvoir transmettre son adresse. Pour cela on utilisera deux paramètres, l'un indiquant l'adresse du bloc où est définie la variable et l'autre le décalage de la variable dans le bloc. Reprendre les questions précédentes lorsque le paramètre `y` est passé par référence.

5 Récursion terminale

On se donne un mini-langage fonctionnel pour la manipulation d'expressions arithmétiques. Un programme de ce langage est composé de la déclaration de fonctions ou de variables suivi d'une expression à évaluer. On remarquera que les variables sont soient locales (paramètres des fonctions) soit globales (déclarées par `val`)

La grammaire est la suivante :

P	≔B E	E	≔`num`
B	≔ϵ	E	≔`id`
B	≔B D	E	≔`id`(F)
D	≔`val id` = E	E	≔`if` E `relop` E `then` E `else` E
D	≔`fun id` (L) = E	E	≔E `op` E
L	≔`id`	F	≔E
L	≔L, `id`	F	≔F, E

`op` (resp. `relop`) est un symbole terminal dont la valeur est l'un des opérateurs arithmétiques $\{+, -, *, /\}$ (resp. $\{<, \leq, =\}$). E représente une expression, D une déclaration de variable globale ou de fonction, L représente une suite d'identificateurs, F une suite d'expressions, B une suite de déclarations.

La valeur d'un programme formé d'une suite de déclarations B et d'une expression E est égale à la valeur de l'expression E évaluée dans l'environnement correspondant à B. Ce langage est compilé vers la machine à pile décrite en cours.
On appelle P le programme suivant :

```
val a = 4
fun square(z) = z*z
fun f(x,y)=square(x)+square(y)
val b = 2
f(a,b)+a
```

1. Donner pour le programme P une manière de stocker les variables globales et les paramètres des fonctions dans la pile. Indiquer pour chaque variable les codes d'accès en lecture ou en écriture.
2. Donner le code compilé correspondant au programme P.
3. On se propose de traduire chaque programme de notre langage vers le code de la machine à pile. On pourra supposer que toutes les déclarations de fonctions ont des noms distincts. On suppose également qu'une analyse a permis de relier chaque utilisation d'une variable à sa déclaration, ce qui nous permet de tester si la variable est locale ou globale et de connaître son décalage.

   La traduction est effectuée à l'aide de fonctions de codage $code_B$, $code_E$, $code_F$ ... pour les principaux non-terminaux de la grammaire. Ces fonctions sont spécifiées par des équations.

Le début de la spécification est le suivant :

$$
\begin{array}{ll}
code_P(B\ E) & = code_B(B)\ code_E(E) \\
code_B(\epsilon) & = [] \\
code_B(B\ D) & = code_B(B)\ code_D(D) \\
code_E(num) & = \texttt{PUSHI}\ num \\
code_E(E_1\ \texttt{op}\ E_2) & = code_E(E_1) code_E(E_2) code_{\texttt{op}}(\texttt{op}) \\
code_{\texttt{op}}(+) & = \texttt{ADD} \\
code_{\texttt{op}}(-) & = \texttt{SUB} \\
code_{\texttt{op}}(*) & = \texttt{MUL} \\
code_{\texttt{op}}(/) & = \texttt{DIV} \\
code_{\texttt{relop}}(<) & = \texttt{INF} \\
code_{\texttt{relop}}(\leq) & = \texttt{INFEQ} \\
code_{\texttt{relop}}(=) & = \texttt{EQUAL}
\end{array}
$$

Compléter cette spécification pour l'ensemble de la grammaire.

4. On suppose dans un premier temps que les fonctions définies ne sont pas récursives. Décrire comment calculer pour chaque programme la taille maximale de la pile utilisée lors de l'exécution de ce programme.
5. On suppose maintenant que les fonctions peuvent être récursives. Que peut-on dire sur la taille de la pile ?
6. La récursion terminale, est un cas particulier de récursion où on peut limiter la taille de la pile.
   On dira qu'un identificateur f est *terminal* dans une expression e si l'une des conditions suivantes est vérifiée :
   – f n'apparaît pas dans e,
   – $e = f(e_1, \ldots, e_n)$ et f n'apparaît dans aucun des e_i,
   – $e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ et f n'apparaît pas dans e_0 et est terminal dans e_1 et e_2.
   Une déclaration : $\texttt{fun}\ f(L) = E$ est récursive terminale si f est terminale dans E.
   On peut par exemple écrire une version récursive terminale de la factorielle :

   ```
   fun fact_it(n,m)=if n=0 them m else fact_it(n-1,n*m)
   fun fact(n)=fact_it(n,1)
   ```

   Décrire une manière de calculer si une déclaration de fonction est récursive terminale.
7. Soit f un identificateur terminal dans une expression e. Montrer que si le calcul de e fait un appel à $f(u_1, \ldots, u_n)$ alors c'est que la valeur de e est la même que la valeur de $f(u_1, \ldots, u_n)$.
8. Si le corps de la fonction récursive terminale $f(x_1, \ldots, x_n)$ fait un appel à $f(u_1, \ldots, u_n)$, on décide d'empiler les valeurs de $u_1, \ldots, u_n$ à la place des arguments $x_1, \ldots, x_n$ et de mettre la valeur de retour de $f(u_1, \ldots, u_n)$ à la place de la valeur de retour $f(x_1, \ldots, x_n)$.
   Comment faut-il modifier la génération de code correspondant à la production $E \coloneqq id(F)$ lorsque l'on compile le corps de la fonction récursive terminale f et que $id = f$.
9. Étendre le calcul de la taille maximale de la pile au cas de déclarations de fonctions récursives terminales compilées de la manière précédente.
10. En utilisant cette stratégie, calculer la taille de la pile nécessaire à l'évaluation de `fact(10)`.

6 Tableaux d'activation

1. Construire l'arbre des déclarations pour le programme suivant:

```
program sort;
var a : array [1..10] of integer; x:integer;
  procedure reada; var i:integer; {...};
  procedure exch (i,j:integer); {x:=a[i];a[i]:=a[j]; a[j]:=x}
  procedure quicks (m,n:integer);
      var k,v : integer;
      function part(y,z:integer) : integer; var i,j:integer;
               {v:=a[y]; ...; exch(i,j)...};
      {if n > m then {k:=part(m,n); quicks(m,k-1); quicks(k+1,n)}}
  {reada;quicks(1,10)}
```

2. Indiquer pour chaque corps de procédure quelles sont les symboles visibles.
3. Donner le niveau de chaque symbole.
4. Donner l'arbre d'activation des appels du programme en supposant que `part(n,m)` renvoie la partie entière de $(n+m)/2$ et en ignorant les appels à la procédure `exch`.
5. Indiquer les différents états de la pile lors de l'exécution des premières instructions du programme.
6. On suppose que chaque tableau d'activation de procédure possède un lien vers le tableau d'activation de la procédure père dans laquelle elle est déclarée. Soit une procédure q appelée dans le corps d'une procédure p. Combien de liens père faut-il suivre à partir de p pour trouver le tableau d'activation du père de q en fonction des niveaux de p et de q ?
7. Montrer que la procédure ancêtre r est forcément la dernière activée de son niveau.
8. En déduire une méthode plus efficace que les liens vers le père pour accéder au tableau d'activation déclarant une variable non locale.

7 Passage de fonctions en paramètres

Lorsqu'une procédure p appelle une fonction f avec un mode de portée lexicale, elle a besoin de deux informations l'adresse du code de la fonction f et l'adresse du tableau d'activation du père de la fonction appelée. Si cette fonction ou procédure appelée est un paramètre de la procédure p, il suffit de considérer que p attend deux informations, l'une est l'adresse de f, l'autre l'adresse du père de f.

Soit le programme:

```
procedure tableau;
var a : array [1..10] of int;
procedure writea; var k : int; { for k:=1 to 10 do write(a[k]) };
function id (i:int) : int; { id:=i };
function carre (i:int) : int; { carre:=i*i }
procedure init (f:int->int);
  var k : int; { for k:=1 to 100 do a[k]:=f(k); writea };
procedure apply (f:int->int,i:int,j:int);
  var k : int; { for k:=i to j do a[k]:=f(a[k]); writea };
{ init(id); apply(carre,2,8) }.
```

1. Donner pour chaque procédure et fonction la forme du tableau d'activation.
2. Donner une séquence d'instructions pour le code compilé de ce programme.

8 Compilation d'un langage orienté objet

(d'après l'examen de janvier 2000)

On se donne un mini-langage orienté objet. Un programme de ce langage est composé de déclarations de classes, suivies de déclarations de variables suivies d'une suite d'instructions. Le but du problème est d'étudier la compilation de ce langage.

Grammaire La grammaire du langage a pour symboles terminaux :

`Cid`	identificateurs de classe
`id`	identificateurs de variable ou méthode
`num`	constantes entières
`op`	opérateurs arithmétiques $\{+, -, *, /, <, \leq, =\}$
`class extends var new if then else fi`	mots clés
`= . := , ( )`	symboles

Les symboles non-terminaux sont : $\{P, C, X, V, M, X, L, E, F, I, J\}$. P représente un programme, C une suite de déclarations de classes, V une suite de déclarations de variables, M une suite de déclarations de méthodes, X une suite de variables associées à des identificateurs de classe (leur *type*), L une valeur gauche (une variable déclarée globalement, le paramètre d'une méthode ou bien une variable d'un objet lui-même désigné par une valeur gauche), E une expression (un entier, une opération arithmétique appliquée à des entiers, un nouvel objet, une variable ou le champ d'un objet), F une suite d'expressions, I une instruction (affectation, conditionnelle, application d'une méthode) et J une suite d'instructions. Les règles sont :

P	≔	C V J
C	≔	ϵ
C	≔	C `class Cid extends Cid` V M
V	≔	ϵ
V	≔	V `var id :=` E
M	≔	ϵ
M	≔	M `method id(X) =` J

L	≔	`id`
L	≔	L`.id`
E	≔	`num`
E	≔	`new Cid`
E	≔	L
E	≔	E `op` E
F	≔	ϵ
F	≔	E
F	≔	F`,` E

I	≔	L`:=` E
I	≔	L`.id(`F`)`
I	≔	`if` E `then` J `else` J `fi`
J	≔	ϵ
J	≔	J I
X	≔	ϵ
X	≔	`id : Cid`
X	≔	X`,` `id : Cid`

Classes prédéfinies On suppose prédéfinis deux identificateurs de classe : `Object` (classe universelle de tous les objets) et `Int` la classe des entiers. On suppose qu'il n'y a pas de variables ni de méthodes définies dans ces classes.

Exemple On appelle P le programme suivant :

```
class Vehicule extends Object
      var pos := 0
      method move(x:Int) = pos:=pos+x
class Voiture extends Vehicule
      var places := 1
      method rejoint(v:Vehicule) =
          if v.pos < pos then v.move(pos-v.pos) else self.move(v.pos-pos)
class Camion extends Vehicule
      var charge := 10
      method move(x:Int) = if x<10 then pos:=pos+x else pos:=pos+10 fi
var c:= new Camion
var v:= new Voiture
v.places:= 4
```

```
v.move(30)
c.move(10)
v.rejoint(c)
```

Visibilité Soit un programme P formé des déclarations de classes C, de variables V et d'instructions J.

Les variables visibles dans les instructions J sont exactement les variables déclarées par V. Les variables d'une classe *Cid* sont composées des variables de la classe qu'elle étend, plus celles qu'elle déclare en propre. Les variables visibles dans une méthode de la classe *Cid* sont toutes les variables de la classe *Cid* ainsi que les paramètres de la méthode.

Exemple la classe `Vehicule` n'a qu'une variable `pos` (elle étend `Object` qui ne contient pas de variable) alors que la classe `Voiture` a deux variables: la variable `pos` héritée de `Vehicule` et la variable `places` déclarée dans `Voiture`. De même la classe `Camion` a deux variables: `pos` héritée de `Vehicule` et `charge` déclarée dans `Camion`.

Table des symboles On suppose que l'analyse de visibilité a été faite, c'est-à-dire que chaque identificateur (de classe, variable, paramètre ou méthode) a une entrée dans une table des symboles. Cette table est accessible à partir de la déclaration de l'identificateur. Chaque utilisation d'un identificateur de classe est reliée à cette entrée de la table de symbole. Une valeur gauche l est formée d'une suite de noms de variables $x_1.x_2 \ldots x_n$, les conditions de visibilité demandent que x_1 soit une variable ou un paramètre de méthode préalablement déclaré. L'utilisation de x_1 peut donc être reliée à sa déclaration via la table des symboles. Par contre, l'interprétation des identificateurs $x_2, \ldots, x_n$ dépend de la classe des objets manipulés, la liaison entre l'utilisation de ces identificateurs et leur déclaration sera donc faite plus tard au moment du typage ou de la compilation. Il en est de même pour les identificateurs de méthode.

C'est à vous de préciser quelles informations doivent être stockées dans la table des symboles pour réaliser les fonctions demandées.

Exemple Pour ajouter une information sur la nature de l'identificateur *id*, on décide que la table des symboles comportera un champ *genre* dont la valeur peut-être `paramètre`, `variable`, `méthode` ou `classe`. Vous pouvez utiliser la notation *id.genre*:=`method` pour enregistrer que l'identificateur *id* est une méthode et tester directement la valeur de *id.genre*=`méthode` pour traiter le cas où *id* est une méthode.

Valeur d'un objet Un objet dans une classe *Cid* est représenté en mémoire en juxtaposant les valeurs des variables de la classe de cet objet dans des cases contiguës de la mémoire. A sa création par la fonction `new` *Cid*, un objet de la classe *Cid* est créé avec les valeurs initiales données aux variables dans la définition de la classe.

Exemple un nouvel objet dans la classe `Vehicule` est représenté par la variable `pos` qui est initialisée à 0; un nouvel objet de la classe `Voiture` est représenté par deux variables : `pos` héritée de la classe `Véhicule` initialisée à 0 et `places` initialisée à 1; un nouvel objet de la classe `Camion` est également composé de deux variables : `pos` initialisée à 0 ainsi que `charge` initialisée à 10.

Valeur d'un programme Un programme P est formé d'une suite de déclarations de classes C, de variables V et d'instructions J. La *valeur* de P est l'ensemble des valeurs des variables V à l'issue de l'exécution des instructions J.

Compilation La compilation des méthodes est analogue à celle des procédures. Une méthode m de paramètres X déclarée dans une classe *Cid* se compile comme une procédure admettant un paramètre supplémentaire `self` : *Cid* que l'on peut choisir de mettre en premier. Ainsi l'invocation $L.m(F)$ est traduite comme l'invocation de la procédure m sur l'objet représenté par la valeur gauche L et les paramètres F. Le passage des paramètres se fait par référence.

Questions

1. Donner un type pour représenter les arbres de syntaxe abstraite de ce langage.
2. **Typage** Les classes permettent d'associer un type à chaque objet. Ainsi `new` *Cid* est un nouvel objet dont le type est la classe *Cid*.

   Dans les langages objet, les types sont ordonnés par une relation d'héritage. Si C_1 et C_2 sont deux identificateurs de classe, alors on dira que $C_1 \leq C_2$ si l'une des conditions suivantes est vérifiée :
   – $C_1 = C_2$;
   – C_1 est une classe déclarée avec la mention `extends` C_2;
   – il existe C_3 un identificateur de classe tel que $C_1 \leq C_3$ et $C_3 \leq C_2$.
   Si une expression E a pour type la classe C_1 et que $C_1 \leq C_2$ alors E a également pour type C_2.

   (a) Étant donné un programme P et deux identificateurs de classe C_1 et C_2, on veut savoir si $C_1 \leq C_2$. Proposer une méthode pour calculer cette information, indiquer la complexité en espace et en temps de votre méthode.

   (b) Une variable est soit un paramètre de méthode qui est déclaré avec son type, soit une variable x déclarée par `var` x `:=` E. Le type de x est alors le type de E. Dans le corps d'une méthode m de la classe *Cid*, `self` est un paramètre implicite, déclaré avec le type *Cid*.
   Une valeur gauche L représente un objet. Si L est de la forme *id* alors c'est une variable ou un paramètre déclaré, relié à sa déclaration dans la table des symboles. Si L est de la forme *L'*`.id` alors L' est une valeur gauche qui représente un objet, soit C_1 la classe de cet objet. *L'*`.id` est correctement typé si *id* est une variable de la classe C_1, auquel cas le type de *L'.id* est égal au type de cette variable.
   E_1 `op` E_2 est une expression bien typée de type `Int` si E_1 et E_2 sont bien typés de type `Int`.
   Question : Donner une méthode pour vérifier qu'une expression est correctement typée et calculer le type de chaque variable déclarée et de chaque expression.

   (c) Indiquer une manière de calculer, pour chaque classe, la taille occupée en mémoire par un objet dans cette classe.

   (d) Les règles suivantes déterminent lorsqu'une instruction I est correcte :
   – L`:=`E est correcte si E est de type C_1, L de type C_2 et $C_1 \leq C_2$;
   – `if` E `then` J_1 `else` J_2 `fi` est correcte si E est une expression de type `Int` (0 représente `faux` et un entier non nul représente `vrai`), et J_1 et J_2 sont des suites d'instructions correctes;
   – L`.id(`F`)` est correcte si :
   L est une valeur gauche qui représente un objet de la classe *Cid*,
   id est une méthode de la classe *Cid* qui attend k arguments de type $C_1, \ldots, C_k$; F représente une liste de k arguments $f_1, \ldots, f_k$ et soit C'_i le type de f_i on a $C'_i \leq C_i$.
   Question : Donner une méthode pour vérifier qu'une séquence d'instructions est bien formée.

   (e) Une déclaration de classe est bien formée, si les déclarations de variables dans la classe sont bien formées et si pour chaque méthode, les déclarations de variables et les instructions de la méthode sont bien formées.
   De plus si $C_1 \leq C_2$, une méthode déclarée dans C_2 peut être redéfinie dans C_1 auquel cas les types des paramètres doivent être les mêmes.
   Question : Donner une méthode pour vérifier qu'une déclaration de classe est bien formée.

3. **Méthodes statiques/ méthodes dynamiques** Une méthode m définie dans une classe C_1 peut être redéfinie dans une classe C_2 telle que $C_2 \leq C_1$. Si on invoque *c.m* il faut

déterminer quel code appliquer, pour cela on s'appuie sur la classe de c. On distingue, la *classe statique* de c qui est la classe déterminée statiquement par le programme (celle que vous avez calculée à la question 2b) de la *classe dynamique* qui est la classe de l'objet correspondant à la valeur de la variable. Par exemple si x est une variable de type C_1 et v un objet de type C_2 alors on peut exécuter l'instruction $x := v$. À la suite de cette instruction, le type dynamique de x est devenu C_2.
Dans le cas statique, l'invocation de $x.m$ correspondant au type déclaré de x soit C_1, dans le cas dynamique, il s'agit du type à l'exécution de x soit C_2.

Question Donner le résultat (valeurs des variables `v` et `c`) de l'évaluation du programme P dans les cas où les méthodes sont compilées de manière statique puis de manière dynamique.

4. **Compilation statique**
   (a) Réécrire le code du programme P dans un langage à la Pascal ne comportant pas de fonctionnalités objet (transformer les classes en record, les méthodes en procédures, introduire des fonctions correspondant à `new`).
   (b) Donner le code compilé correspondant au programme P dans le cas où les méthodes sont toutes statiques. On utilisera le code intermédiaire de la machine à pile vue en cours.
5. **Compilation dynamique** Dans le cas de la compilation dynamique, le label du code à exécuter ne peut être résolu qu'au moment de l'exécution. On choisit donc de stocker avec chaque objet en plus des valeurs des variables, les labels des différentes méthodes que cet objet peut invoquer.
   (a) Est-il en général possible de trouver un programme Pascal équivalent au programme objet avec méthodes dynamiques ?
   (b) Quelles instructions faut-il ajouter à la machine pour compiler le code correspondant ?
   (c) Donner le code compilé correspondant au programme P.

9 Compilation de code Java

Soit le programme Java suivant :

```
class Vehicule
{ int pos;
    Vehicule () {pos = 0;}
    void move (int x) { pos = pos+x; }
}

class Voiture extends Vehicule
{ int places;
    Voiture () {places = 1;}
    void rejoint (Vehicule v)
    { if (v.pos < pos) v.move(pos-v.pos); else this.move(v.pos-pos);}
}

class Camion extends Vehicule
{int charge;
    Camion () {charge = 10; }
    void move (int x) { if (x<10) pos = pos+x; else pos = pos+10; }
}

class Main {
```

```
    static Camion c = new Camion ();
    static Voiture v = new Voiture ();
    static void main(String [] argv) {
        v.places = 4; v.move(30); c.move(10); v.rejoint(c);
        System.out.println(v.pos);
        System.out.println(c.pos); }
}
```

1. Quel est le résultat de ce programme ?
2. On décide de représenter le descripteur d'une classe C avec un pointeur sur le descripteur de la classe que C étend et les pointeurs sur les codes des méthodes non statiques appliquables aux objets de la classe C. Proposer un code de la machine abstraite qui stocke dans la mémoire globale les descripteurs des classes de ce programme. Indiquer l'état de la pile à l'issue de l'exécution de ce code. On utilisera les instructions de la machine à pile vue en cours augmentée de l'instruction `ALLOC` n qui permet d'allouer un bloc de taille n dans le tas et d'empiler l'adresse obtenue sur la pile.
3. Expliquer la représentation des objets pour chacune des classes non statiques. Proposer un code qui traduise l'effet des instructions `new` sur chacune de ces classes. La commande `DUP` n permet de dupliquer les n valeurs en sommet de pile.
4. Proposer un code pour la fonction `instanceof` entre un objet et une classe. Pour cela on précisera l'algorithme utilisé ainsi que le tableau d'activation de la fonction `instanceof`.
5. Comment traduit-on les instructions de **cast** lorsque ceux-ci mettent en jeu des types numériques.
6. Proposer un format de tableau d'activation et donner le code pour les différentes méthodes dynamiques du programme Java.
7. Donner le codage des variables et méthodes statiques de la classe `Main`.
8. Préciser l'organisation de la table des symboles qui vous sera utile pour engendrer ce code de manière générale. Donner le schéma général de compilation pour les instructions d'affectation à un champ d'un objet : $e_1.c = e_2$, l'appel de méthode $e.m(e_1, \ldots, e_n)$.
9. Dans cette question, nous nous intéressons a l'optimisation d'un appel de méthode. Notons que même dans le cas d'un langage avec héritage simple, un appel de méthode est plus couteux qu'un appel de fonction. Il est nécessaire de récuperer l'adresse du descripteur de classe de l'objet puis l'adresse de la méthode dans cette classe avant d'effectuer l'appel.
   Pour un appel de méthode *o.m()* ou o est de classe C, l'analyse de la hierarchie des classes (donnée par la relation d'héritage) peut etre utilisée pour déterminer les sous-classes de C redéfinissant C_m. S'il n'existe aucune redéfinition de C_m alors l'instance de la méthode est nécessairement C_m. Donner le code de la compilation de l'appel de méthode dans ce cas.

10 Génération de code pour langages objets

(Examen septembre 2001)

1. Expliquer en quelques lignes la différence entre appel de méthode statique et appel de méthode dynamique dans les langages à objets. Donner un exemple.
2. Expliquer les grandes lignes du schéma de compilation des appels de méthodes dynamique et statique. Quel type d'appel de méthode est le plus efficace ?

11 Compilation d'un langage avec tableaux dynamiques

(Examen janvier 2001)

Ce problème étudie la compilation d'un mini-langage fonctionnel permettant de manipuler des tableaux de taille arbitraire contenant des données qui peuvent être elles-mêmes des tableaux.

11.1 Syntaxe du langage

Un programme du langage est composé d'une suite de déclarations.

Déclarations Une *déclaration de variable* s'écrit : **let** *id* = *exp* avec *id* le nom de la variable et *exp* sa valeur.

Une *déclaration de fonction* s'écrit : **let** *id* $(id_1, \ldots, id_n) = exp$ avec *id* le nom de la procédure, id_i le nom du i-ème paramètre, *exp* une expression représentant le corps de la fonction et pouvant faire référence à *id*.

Expressions Une *expression* peut être :

- une variable, déclarée dans un **let** ou bien comme paramètre d'une fonction.
- une valeur entière ou booléenne (`true` ou `false`).
- une expression exp_1 `op` exp_2 formée d'un opérateur binaire prédéfini `op` appliqué à deux expressions exp_1 et exp_2 avec `op` $\in \{+, -, =, <, \leq\}$
- un appel de fonction $id(exp_1, \ldots, exp_n)$
- une expression conditionnelle : **if** *exp* **then** exp_1 **else** exp_2.
- une expression avec déclaration locale *decl* **in** *exp* où *decl* est une déclaration de variable ou de fonction éventuellement récursive avec la même syntaxe que les déclarations globales.

Fonctions prédéfinies On suppose que les fonctions suivantes permettant la manipulation des tableaux sont prédéfinies :

`create` qui étant donné un entier n et une expression x, crée un tableau de taille n initialisé par la valeur de x;

`length` qui étant donné un tableau renvoie sa longueur;

`get` qui étant donné un tableau t et un entier i compris entre 1 et la longueur de t, renvoie la valeur $t[i]$ du tableau t à l'indice i;

`set` qui étant donné un tableau t, un entier i compris entre 1 et la longueur de t, et une expression x, affecte la valeur de x à l'emplacement d'indice i du tableau et renvoie le tableau t ainsi modifié.

11.2 Typage

Les fonctions peuvent prendre des tableaux en arguments et renvoyer des tableaux. Les données manipulées sont des entiers ou bien des tableaux formés d'objets quelconques. Les fonctions peuvent être polymorphes, c'est-à-dire s'appliquer à des tableaux contenant des objets arbitraires.

Les *types* du langage sont soit le type de base des entiers `int`, soit le type des booléens `bool`, soit un type tableau contenant des objets de type τ, noté τ `array`.

Pour traiter le polymorphisme, on introduit les *schémas de type* qui peuvent contenir des variables de type qui représentent des types arbitraires. Un schéma de type est donc soit `int`, soit `bool`, soit une variable de type (notée $\alpha, \beta, \ldots$), soit un schéma de type tableau noté σ `array` avec σ un schéma de type.

Un schéma de type représente un ensemble de types obtenus en remplaçant les variables de type par des types quelconques. Ainsi le schéma de type α représente tous les types, le schéma de type α `array` représente tous les types de la forme τ `array` avec τ un type (ie `int array`, `bool array`, (`int array`) `array`, ((`int array`) `array`) `array` ...)

À toute fonction est associée un *profil* de la forme $\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$ où σ_i sont des schémas de type.

On peut associer aux opérations primitives les profils suivants :

- `create` : `int` $\times \alpha \rightarrow \alpha$ `array`
- `length` : α `array` $\rightarrow$ `int`
- `get` : α `array` $\times$ `int` $\rightarrow \alpha$
- `set` : α `array` $\times$ `int` $\times \alpha \rightarrow \alpha$ `array`

1. Dire ce que font les fonctions suivantes et donner leur profil.

```
let f(x,y)=x
let get_matrix(m,i,j) = get(get(m,i),j)
let set_matrix(m,i,j,v) = set(get(m,i),j,v)
```

2. Écrire une fonction `add_array` qui prend en argument deux tableaux d'entiers x et y de même taille n et renvoie un nouveau tableau z de taille n tel que pour tout i compris entre 1 et n on ait $z[i] = x[i] + y[i]$.

11.3 Génération de code

Le langage est compilé vers le code d'une machine à pile dont les instructions sont rappelées à la fin de ce document.

Les tableaux sont alloués dans la zone du tas. La valeur d'un tableau sera l'adresse du bloc dans lequel le tableau est stocké. Un tableau n'est jamais recopié lors d'un appel de fonction, seule l'adresse est passée en argument. La seule fonction qui alloue de la place pour un tableau dans le tas est la fonction `create`.

Par rapport à la machine vue en cours, on suppose que l'on dispose de la commande suivante :

`ALLOC` dépile un entier n, réserve un bloc de $n+1$ mots dans le tas et renvoie l'adresse *addr* du bas du bloc au sommet de la pile (les élements du bloc seront donc placés aux adresses *addr* à $addr + n$)

1. Peut-on connaître statiquement (à la compilation) la taille du tableau correspondant à la valeur d'une expression quelconque du langage, en particulier pour l'expression correspondant au corps d'une fonction ?
2. Proposer une manière de représenter les tableaux qui permette que tous les objets manipulés dans la pile (entiers ou tableaux) aient la même taille et d'implanter la fonction `length`.
3. On veut coder les opérations primitives (`length`, `get`, `set` et `create`). Expliquer l'organisation du tableau d'activation créé par l'appel de ces fonctions et donner le code de la machine à pile correspondant au corps de chacune de ces procédures
4. Soit le programme :

```
let x = create(2,0)
let y = x
let x' = set(x,1,2)
let v = get(y,1)
```

   Indiquer l'état de la pile et du tas après l'exécution de chaque déclaration.
5. Donner le code machine correspondant à la fonction `add_array` définie précédemment.
6. L'étudiant Arthur doit implanter une fonction d'addition de trois tableaux de même taille, il décide de réutiliser la fonction `add_array` définie précédemment et procède de la manière suivante :

```
let add3_array(x,y,z) = let t = add_array(x,y) in add_array(t,z)
```

   Quel défaut présente la solution choisie ? Quelles solutions pouvez-vous proposer pour y remédier ?
7. On veut étendre le langage pour pouvoir spécifier dans une déclaration de fonction qu'un paramètre (de type tableau) est passé par valeur, c'est-à-dire qu'un nouveau tableau est créé au moment de chaque appel. On introduit pour cela la déclaration **let** f (**val** x) = *exp*.

Lors de l'appel de $f(a)$, les modifications apportées aux éléments de x dans l'expression *exp* n'auront pas d'incidence sur les éléments de a.

Montrer que l'on peut effectuer cette extension du langage sans modifier le compilateur vers la machine à pile, en transformant simplement une déclaration de fonction avec appel par valeur en une déclaration de fonction avec appel par référence ayant le même comportement.

Machine à pile

Les instructions de cette machine à pile sont analogues à celles décrites dans le cours. Le registre *gp* indique le bas de la pile, le registre *fp* indique le pointeur de référence des variables locales, le registre *sp* indique la première case libre de la pile. Des instructions `LABEL` peuvent être introduites dans le code, elles ne modifient pas l'état de la pile. "Empiler x" signifie ajouter x au sommet de la pile (*sp* est augmenté de 1). "Dépiler x" signifie retirer la valeur au sommet de la pile qui sera notée x (*sp* est diminué de 1).

`PUSHI` n	empile la constante entière n.
`ADD` (resp. `SUB`)	dépile y puis x et empile $x+y$ (resp. $x-y$)
`EQ` (resp. `INF`,`INFEQ`)	dépile y puis x et empile 1 si $x=y$ (resp. $x<y$, $x\leq y$) et 0 sinon)
`PUSHL` n	empile la valeur située n cases au dessus de *fp*.
`STOREL` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *fp*.
`LOAD` n	dépile une adresse a et empile la valeur située n cases au dessus de a.
`STORE` n	dépile une valeur x, une adresse a et stocke x à l'adresse située n cases au dessus de a.
`LOADN`	dépile un entier n et une adresse a, empile la valeur située n cases au dessus de a.
`STOREN`	dépile une valeur x, un entier n et une adresse a, et stocke x à l'adresse située n cases au dessus de a.
`PUSHN` n	empile n valeurs nulles sur la pile.
`POP` n	dépile n valeurs de la pile.
`JZ` l	dépile x, si $x=0$ le pointeur de code saute à l'instruction `LABEL` l.
`JUMP` l	saute à l'instruction `LABEL` l.
`CALL` f	sauve la valeur courante de *fp* et le pointeur d'instructions, affecte à *fp* la valeur de la première case libre de la pile, saute à l'instruction `LABEL` f.
`RETURN`	dépile toutes les valeurs au-dessus de *fp*, restaure l'ancienne valeur de *fp*, reprend l'exécution à l'instruction suivante la dernière instruction `CALL`.

12 Structures, passage par valeur ou par référence

On considère un langage dans lequel on manipule des entiers, des booléens et des paires d'objets qui seront représentés par des éléments consécutifs de la mémoire.

Les types de base sont les entiers et les booléens et dans lequel on peut former des types produits de deux types.

Programmes Les types sont donnés par la grammaire :

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau * \tau$$

Les expressions sont données par la grammaire :

$$e ::= \texttt{cte\_int} \mid \texttt{cte\_bool} \mid \texttt{id} \mid (e,e) \mid e.1 \mid e.2 \mid e\ \texttt{op}\ e$$

Parmi les expressions, on distingue les valeurs gauches :

$$g ::= \texttt{id} \mid g.1 \mid g.2$$

Les instructions sont formées d'affectations, d'appels de procédures et peuvent être mises en séquence.

$$i ::= g = e \mid i; i \mid \texttt{id}(e, \ldots, e)$$

Un programme est donné comme une suite de déclarations de procédures de la forme :

$$\texttt{procedure id}(\mathit{params})\{\mathit{vars} \mid i\}$$

Les paramètres peuvent être déclarés par valeur : $\texttt{id} : \tau$ ou bien par référence $\texttt{var id} : \tau$. Les variables locales sont déclarées avec leur type.

Arbres de syntaxe abstraite Les arbres de syntaxe abstraite sont déclarés ainsi :

```
type typ = Int | Bool | Prod of typ * typ
type expr =  Var of ident
           | Binop of binop * expr * expr
           | CteBool of bool
           | CteInt of int
           | Pair of expr * expr
           | Proj of bool * expr
type instr =  Aff of expr * expr
           | Seq of instr * instr
           | Proc of ident * expr list
type pass = Ref | Val
type proc = ident * (pass * ident * typ) list * (ident * typ) list * expr
```

On représente $e.1$ (resp. $e.2$) par `Proj(true,e)` (resp. `Proj(false,e)`). Dans les affectations, on représente les valeurs gauches par des expressions.

1. Donner le code à engendrer pour les programmes suivants :

   ```
   procedure p (var x:int; y:int) { x = x+y }
   procedure q () { z:int | z=0; p(z,3) }
   procedure r (a:int,b:int) { p:int*int | p.1=a;p.2=b }
   ```

2. On suppose que la table des symboles contient pour chaque variable son type et des informations utiles à la compilation (de type `gen_var`) et pour chaque procédure la liste des types des arguments ainsi que des informations utiles à la compilation (de type `gen_fun`). On dispose des fonctions suivantes pour accéder aux valeurs dans la table des symboles ou bien les mettre à jour.

   ```
   val type_var : string -> typ
   val gen_var : string -> gen_var
   val set_gen_var : string -> gen_var -> unit
   val prof_fun : string -> typ list
   val gen_fun : string -> gen_fun
   val set_gen_fun : string -> gen_fun -> unit
   ```

   Préciser les types `gen_var` et `gen_fun` afin d'y stocker les décalages des variables, et autres informations utiles à la compilation.
   Donner le code d'une fonction qui traite les déclarations de procédures et met à jour ces informations.

3. On suppose que la table des symboles est à jour et que les programmes sont bien formés. Écrire une fonction `typeof` qui calcule (sans vérifier) le type d'une expression.

4. On choisit de calculer les expressions sur la pile et de représenter les données dans les types produits par des éléments consécutifs sur la pile. Donner le code pour l'expression (e_1, e_2) et celle pour l'expression $e.1$ ainsi que le code pour une variable (on distinguera le cas où la variable est passée par valeur ou par référence).

5. Quel problème pose la compilation de l'expression $e.2$? Montrer comment compiler la valeur droite d'une expression qui est une valeur gauche (des projections appliquées à une variable).
6. Donner le code d'une affectation $g := e$
7. Quel problème supplémentaire se pose si on ajoute dans le langage des tableaux et des accès dans les tableaux qui peuvent apparaître dans des valeurs gauches.
8. On se place maintenant dans un langage où les fonctions de projections ne peuvent être utilisées que dans des expressions qui sont des valeurs gauche et où l'utilisation de paires explicites ne se fait que dans des affectations $x = (e_1, e_2)$. On choisit de représenter une valeur produit par l'adresse à partir de laquelle le produit est stocké. Les valeurs de base (entier, booleen) seront toujours représentées par des valeurs ordinaires.
   (a) Donner le code pour les expressions $e.1$ et $e.2$.
   (b) Ecrire le code machine d'une fonction`copy` qui étant données deux adresses x et y et une taille n sur la pile, recopie n cases à partir de y vers x.
   (c) En déduire le code de l'affectation $e_1 = e_2$, on choisit de se placer dans un cadre général où le résultat de la compilation d'une valeur gauche a pour effet de mettre l'adresse de la case mémoire correspondante sur la pile.

13 Langage avec valeurs de paramètres par défaut

(Examen septembre 2001)

La grammaire du langage a pour symboles terminaux :

$$\{\texttt{let}, \texttt{type}, \texttt{id}, \texttt{num}, \texttt{true}, \texttt{false}, \texttt{op}, \texttt{:=}, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{,}, \texttt{(}, \texttt{)}\}$$

`id` représente un identificateur correspondant à une variable ou une fonction, `num` représente une constante entière, `op` est un des opérateurs arithmétiques $\{+, -, *, /, =, <, >\}$, `type` représente le type d'une expression qui peut être l'un des deux mots clés `int` ou `bool`.

Les expressions représentent des entiers ou des booléens. Elles sont engendrées par la grammaire suivante dans laquelle F représente une suite d'expressions (éventuellement vide) séparées par des virgules :

E := `num`	E := E `op` E	F := ϵ
E := `true`	E := `(` E `)`	F := F_1
E := `false`	E := `id(` F `)`	F_1 := E
E := `id`	E := `if` E `then` E `else` E	F_1 := F_1 `,` E

Un programme dans ce langage est composé d'une suite de déclarations qui peuvent être soit des déclarations de fonctions soit l'association d'un nom à une expression. Par exemple :

```
let square (x:int) : int = x * x
let f (x:int,y:int) : int := square(x) + square(y)
let x := 3
let y := 4
let z := f (x,y)
```

La particularité de ce langage est de permettre de donner des valeurs par défaut à certains paramètres des fonctions. Une valeur par défaut peut être n'importe quelle expression. Au moment de l'appel, il n'est pas nécessaire de fournir une valeur pour un tel paramètre, à défaut, la valeur associée au paramètre au moment de la déclaration de la fonction sera utilisée. Ainsi on peut écrire :

```
let g (x:int,y:int,k=1) : int := k * square(x) + square(y)
let t := g(x,y)
let u := g(x,y,2)
```

L'identificateur t aura pour valeur $1 * \texttt{square}(3) + \texttt{square}(4)$ c'est-à-dire 25 tandis que u aura pour valeur $2 * \texttt{square}(3) + \texttt{square}(4)$ c'est-à-dire 50 .

On choisit dans la déclaration des paramètres de regrouper tous les paramètres sans valeur par défaut d'abord puis de mettre les paramètres avec valeurs par défaut ensuite.

1. Compléter la grammaire des expressions pour donner la grammaire complète des programmes de ce langage.
2. Compléter les déclarations de type suivantes (`expr` et `prog`) pour représenter les arbres de syntaxe abstraite de ce langage.

   type oper = Div | Plus | Mult | Eq | Minus | Lt | Gt
   type typ = Int | Bool
   type expr = Ident of string | IntCte of int | BoolCte of bool | Oper of oper * expr * expr
   | ...
   type prog = ...
3. Proposer une structure pour la table des symboles des fonctions et des variables qui permette de vérifier qu'un programme est bien formé, c'est-à-dire qu'il respecte les règles usuelles de portée et de typage :
   - toute variable utilisée dans une expression doit au préalable avoir été déclarée.
   - Les paramètres d'une fonction ne sont visibles que dans le corps de la fonction, les fonctions peuvent être récursives
   - Les opérateurs s'appliquent à des entiers, sauf `=` qui peut s'appliquer à deux booléens.
   - Dans une expression conditionnelle, la condition a pour type un booléen et les deux branches ont le même type.
4. Décrire un programme permettant de vérifier qu'un programme est correctement formé. Il n'est pas nécessaire de détailler les opérations de manipulation de la table des symboles, il suffit de donner leur spécification.
5. On souhaite engendrer du code assembleur pour calculer les valeurs de chaque variable introduite dans le programme. Expliquer de quelle manière on peut traiter les arguments donnés par défaut : détailler la déclaration d'une fonction avec paramètres ayant une valeur par défaut ainsi que l'appel d'une telle fonction.
6. Que faut-il changer au mode de compilation si on autorise les valeurs par défaut à faire référence aux paramètres de la fonction comme dans l'exemple :

   ```
   let h (x:int,y:int,k=x) = k * square(x) + square(y)
   ```

14 Génération de code : `break`, `continue`

(Septembre 2002)

On considère un mini-langage impératif *BC* comportant des conditionnelles et des boucles et seulement des variables globales. Ce langage comporte de plus deux instructions `break` et `continue` qui permettent d'interrompre l'exécution du corps d'une boucle. Lorsque dans le corps d'une boucle, on rencontre l'instruction `break` alors l'exécution se poursuit à la fin de la boucle, lorsque l'on rencontre l'instruction `continue` alors l'exécution reprend au test de la boucle. Lorsque plusieurs boucles sont imbriquées, les instructions `break` et `continue` permettent de sortir de la première boucle englobante.

On cherche à compiler le langage *BC* vers les instructions de la machine à pile vues en cours et rappelées à la fin de l'énoncé.

Les expressions du langage BC sont soit des constantes entières, soit des variables (globales) soit une opération binaire (arithmétique ou test d'égalité) appliquée à deux expressions, on convient que les booléens `true` et `false` sont représentés respectivement par les entiers 1 et 0.

Une instruction de BC peut être soit une affectation $x := e$, soit une conditionnelle `if` e `then` s_1 `else` s_2 `end` ou bien `if` e `then` s `end`, soit un bloc formé d'une liste d'instructions $\{s_1; \ldots; s_n\}$ soit une boucle `while` e `do` s `end`, soit une instruction d'échappement `break` ou `continue`.

Les types CAML suivants permettent de représenter les arbres de syntaxe abstraite de ce langage (les variables sont directement représentées par leur décalage par rapport à la base de la pile):

```
type op = Plus | Mult | Sub | Mod | Eq
type var = int
type expr = Const of int | Var of var | Op of op * expr * expr
type stat =   Aff of var * expr
            | If of expr * stat * stat
            | Block of stat list
            | While of expr * stat
            | Break | Continue
```

1. Écrire une fonction de compilation des instructions de BC vers la machine à pile dans le cas où il n'y a pas d'instruction `break` ou `continue`.
2. Soit le programme de BC :

   ```
   { p:=2;
     while true do
         if p=n then { r:=1; break } end;
         if n mod p = 0 then { r:=0; break }
         else p:=p+1 end
     end }
   ```

   Donner des instructions de la machines à pile correspondant au code compilé de cette fonction.
3. Écrire une fonction de compilation dans le cas général où les corps de boucle peuvent mentionner les instructions `break` et `continue`.
4. Dans un langage comme Java les instructions `break` ou `continue` peuvent mentionner un label qui a été introduit en tête d'une instruction `while`. Une instruction `break` ou `continue` sans label sort de la première boucle `while` englobante tandis qu'une instruction `break` (resp. `continue`) avec un label l sort de (resp. réitère) la boucle `while` portant le label l.

   Les constructeurs `While`, `Break` et `Continue` du type `stat` sont modifiés de la manière suivante :

   ```
   type label = None | Tag of string
   type stat = ...
               While of label * expr * stat
             | Break of label | Continue of label
   ```

   Modifier la fonction de compilation des instructions de BC pour traiter ce cas.

Machine à pile

Les instructions de cette machine à pile sont analogues à celles décrites dans le cours, on a juste ajouté une opération `MOD` qui calcule directement le reste de la division entière de deux nombres : On a x **mod** $y = x - (x/y) * y$.

15 Compilation des expressions `switch`

(septembre 2003)

En C ou en Java, la structure de controle `switch` permet d'effectuer un branchement à choix multiples. La syntaxe d'une instruction `switch` est

switch (*expr*) {*opt-label-instr-list*}

où *expr* est une expression valide et le corps *opt-label-instr-list* du `switch` est une suite d'instructions chacune pouvant être préfixée par un label de choix. Un label de choix est soit `default`: soit un ou plusieurs labels de la forme `case` *cste*: avec *cste* une constante du langage.

Un exemple de programme utilisant cette instruction est donné dans la figure 1.

```
#define EOF -1
main()
{int c, nwhite, nother;
 nwhite = nother = 0;
 while ((c=getchar()) != EOF)
  switch (c) {
   case ' ':
   case '\n':
   case '\t':
     nwhite++; break;
   default:
    nother++; break;
   }
printf("white space = %d, other = %d\n",nwhite,nother);
}
```

FIG. 1 – Exemple de programme utilisant l'instruction **switch**

L'instruction **switch** doit vérifier que toutes les constantes apparaissant dans les labels **case** sont distinctes et qu'il y a au plus un label **default**. Dans la suite on supposera que ces conditions sont toujours vérifiées.

La sémantique de l'instruction **switch** (*e*) {*li*} est d'évaluer l'expression *e* en une valeur *v*, de chercher dans la suite d'instructions avec labels *li* s'il y a une étiquette **case** *c* telle que la valeur de la constante *c* est aussi *v*. Trois cas peuvent se produire :

- Si un label **case** *c* de valeur *v* existe, les instructions de *li* sont exécutées à partir de l'instruction comportant ce label,
- s'il n'existe pas de label de valeur *v* mais qu'il y a un label **default** alors les instructions de *li* sont exécutées à partir de l'instruction comportant le label **default**,
- s'il n'y a pas de label de valeur *v* ni de label **default** alors aucune instruction n'est exécutée.

Dans tous les cas l'instruction **break** apparaissant dans *li* provoque la sortie de l'instruction `switch`.

1. On suppose que la grammaire du langage est partiellement écrite avec des non-terminaux *cste*, *expr* et *instr* qui reconnaissent respectivement les expressions constantes, les expressions du langage, et les instructions et qui comportent au moins les règles :

expr	:= `ident`	*cste*	:= `'char'`
expr	:= *cste*	*instr*	:= **break**;
expr	:= `ident++`	*instr*	:= *expr*;

avec `char` et `ident` des terminaux correspondant aux unités lexicales des caractères et des identificateurs.

Compléter la grammaire pour accepter les instructions **switch**.

2. Donner l'arbre de dérivation syntaxique de l'instruction :

```
switch (c) {
 case ' ':
 case '\n':
 case '\t':
   nwhite++; break;
 default:
   nother++; break;
 }
```

3. On suppose définis des type de données `asa_cste`, `asa_expr` et `asa_instr` pour représenter respectivement les arbres de syntaxe abstraite des constantes, expressions et instructions. La figure 2 donne un sous-ensemble des constructeurs des arbres de syntaxe abstraite avec leurs types.

Char	: `char` → `asa_cste`
Var	: `string` → `asa_expr`
Cste	: `asa_cste` → `asa_expr`
Incr	: `string` → `asa_expr`
Expr	: `asa_expr` → `asa_instr`
Break	: `asa_instr`

FIG. 2 – Constructeurs pour les arbres de syntaxe abstraite

   (a) Proposer un type pour le constructeur de l'arbre de syntaxe abstraite `asa_instr` correspondant à l'instruction `switch`.

   (b) Donner l'expression correspondant à l'arbre de syntaxe abstraite de l'instruction de la question 2.

4. On suppose donné une fonction *val_ cste* de type `asa_cste` → `int` qui calcule la valeur (entière) d'une expression constante. Pour un caractère, il s'agit de son code ASCII. On rappelle que les codes ASCII de $'\backslash t'$, $'\backslash n'$ et $'\ '$ sont respectivement 9, 10 et 32.
   On se propose de compiler notre langage vers les instructions de la machine à pile du cours dont un rappel est donné en 21. On construit donc des fonctions *compil_ expr* et *compil_ instr* qui produisent à partir des arbres de syntaxe abstraite de notre langage initial, les instructions correspondantes de la machine à pile. Dans le cas d'une expression, sa valeur à la fin de l'exécution se trouve au sommet de la pile.
   Donner le schéma de compilation général pour une instruction `switch` en supposant que les fonctions *compil_ expr* et *compil_ instr* sont déjà définies pour les autres cas.

5. Donner la suite d'instructions de la machine à pile pour l'instruction de la question 2.

6. On ajoute maintenant à notre machine une instruction de saut indéxé `JUMPI` qui prend en argument un label l et effectue un saut du pointeur de code *pc* à l'adresse $l + i$ si i est la valeur du sommet de la pile (qui est dépilé).
   L'idée est de construire pour chaque instruction `switch` un tableau correspondant à l'éventail des cas possibles du `switch`.
   Dans le cas de l'exemple on a trois cas à tester : les caractères ' ', '\t' et '\n' correspondant aux valeurs numériques de $32, 9$ et 10. Il suffit donc de garder une table de choix pour les valeurs numériques entre 9 et 32. Dans cette table de taille $32 - 9 + 1 = 24$, les cases 0, 1 et 23 correspondant aux valeurs 9, 10 et 32 contiendront l'instruction `JUMP` l si l est le label de l'instruction qui suit ces cas; les autres cases contiendront l'instruction `JUMP` *def* si *def* est le label de l'instruction par défaut.

On suppose que chaque instruction **switch** est prétraitée, et qu'on lui associe un label *choix* qui indique la place du tableau de choix, deux entiers *min*, *max* qui donnent la valeur minimum et la valeur maximum apparaissant dans les **case** ainsi qu'une étiquette *fin* pour marquer la fin du **switch** et une étiquette *def* pour marquer les instructions de la clause par défaut (cette étiquette est la même que l'étiquette *fin* s'il n'y a pas de clause par défaut) et finalement une liste d'étiquettes *lcases* pour chaque instruction préfixée d'un **case**.

Dans le cas de notre exemple, le programme compilé comporterait une table d'initialisation décrite dans la figure 3.

choix:
`JUMP` l — indice 0 correspondant au caractère '\t' de valeur 9
`JUMP` l — indice 1 correspondant au caractère '\n' de valeur 10
`JUMP` *def* — indice 2 correspondant au caractère de valeur 11
`JUMP` *def* — indice 3 correspondant au caractère de valeur 12
`JUMP` *def* — indice 4 correspondant au caractère de valeur 13
`JUMP` *def* — indice 5 ...
`JUMP` *def* — indice 6 ...
`JUMP` *def* — indice 7 ...
`JUMP` *def* — indice 8 ...
`JUMP` *def* — indice 9 ...
`JUMP` *def* —indice 10 ...
`JUMP` *def* —indice 11 ...
`JUMP` *def* —indice 12 ...
`JUMP` *def* —indice 13 ...
`JUMP` *def* —indice 14 ...
`JUMP` *def* —indice 15 ...
`JUMP` *def* —indice 16 ...
`JUMP` *def* —indice 17 ...
`JUMP` *def* —indice 18 ...
`JUMP` *def* —indice 19 ...
`JUMP` *def* —indice 20 ...
`JUMP` *def* —indice 21 ...
`JUMP` *def* —indice 22 ...
`JUMP` l —indice 23 correspondant au caractère ' ' de valeur 32

FIG. 3 – Table d'initialisation du **switch**

À l'endroit d'une instruction **switch** (e) { li }, on insère le code pour evaluer l'expression *e*, le code pour effectuer le branchement indexé aproprié en fonction de la valeur de *e* puis enfin les instructions du corps du **switch** avec les étiquettes correspondantes. Dans le cas de notre exemple, on obtient :

—*code pour la partie expression du* ***switch***
`PUSHG` (*decalage* c)
—*code pour effectuer le branchement aproprié en fonction de la valeur de c en utilisant la table stockée en choix et l'instruction* `JUMPI`
... À COMPLÉTER...
l:
*code pour les instructions nbwhite++;****break;***
def:
*code pour les instructions nbother++;****break;***
fin:

(a) Compléter le code pour effectuer le branchement dans le cas de l'exemple.

(b) Donner le schéma général pour cette partie du code en supposant données les valeurs *min* et *max* ainsi que les étiquettes *choix* et *def*.

(c) Quels sont les avantages et les inconvénients de cette seconde méthode de compilation par rapport à la méthode qui n'utilise que les instructions de branchement `JZ` et `JUMP` ?

7. L'étudiant Gus doit écrire un programme interactif qui comprend en particulier les instructions `Find`, `Load`, `Write` et `Quit`. Il choisit d'associer à chaque instruction un caractère (la première lettre) et écrit son programme sous la forme :

   switch (c) {
   case 'f':...
   case 'l':...
   case 'w':...
   case 'q':...
   default:...
   }

   Son collègue, qui préfère utiliser CAML, commence par définir un type de données `command` puis utilise du filtrage :

   type `command` = `Find` | `Load` | `Save` | `Quit`
   match (c) **with**
   | `Find` → ...
   | `Load` → ...
   | `Save` → ...
   | `Quit` → ...

   Sachant que CAML associe à chaque constructeur un numéro et compile le filtrage à l'aide d'une instruction analogue à **switch**, quels avantages le compilateur peut-il tirer de cette deuxième approche ?

16 Allocation de registres

Soit le programme écrit en langage intermédiaire :

```
y:=1
m:=x
e:=n
loop:
if e=0 goto fin:
t:=y mod 2
if t=0 goto pair:
y:= m*y
pair:
m:=m * m
e:=e div 2
goto loop:
fin:
print y
```

1. Donner en chaque point de programme, l'ensemble des variables vivantes (variables qui seront lues avant d'être affectées).
2. Construire le graphe d'interférence (graphe dont les sommets sont les variables et où il y a une arête entre x et y si elles sont simultanément vivantes).

3. Proposer un coloriage du graphe pour mettre les variables dans 4 registres.
4. Montrer qu'il n'est pas possible d'utiliser uniquement 3 registres.
5. On se propose donc de stocker la variable y en mémoire à l'adresse Y. Proposer une nouvelle version du programme, dans laquelle chaque accès à la variable y passe par une variable intermédiaire et l'accès à la mémoire. On utilisera uniquement des instructions

   `mem[Y]:=n, mem[Y]:=x, x:=mem[Y]`

   pour mettre en mémoire une constante, la valeur d'une variable ou récupérer dans une variable une valeur en mémoire.
6. Donner les variables vivantes pour ce nouveau programme.
7. Proposer une allocation des variables utilisant uniquement 3 registres.

17 Allocation de variables locales

(Examen janvier 04)

On étudie un langage pour des expressions qui contiennent des définitions locales de variables. Ce langage est décrit par la grammaire suivante dans laquelle `cte` est un token correspondant à une valeur entière $(1, 2, \ldots)$, `ident` est un token correspondant à un identificateur qui peut représenter une variable locale ou bien une fonction prédéfinie et finalement `let` et `in` sont des mots clés et `=`, `(`, `,` et `)` sont des symboles. E est un non-terminal permettant de dériver les expressions du langage.

$$E ::= \texttt{cte} \mid \texttt{ident} \mid \texttt{let ident =} E \texttt{ in } E \mid \texttt{ident(}E\texttt{,}E\texttt{)} \mid \texttt{(}E\texttt{)}$$

Un type CAML possible pour les arbres de syntaxe abstraite de ce langage est

```
type ast = Cte of int | Var of string
         | Let of string * ast * ast | Fun of string * ast * ast
```

Dans une expression `let` x `=` e_1 `in` e_2, x est une nouvelle variable qui peut être utilisée uniquement dans l'expression e_2. L'évaluation de l'expression complète consiste à évaluer e_1, à stocker la valeur obtenue dans une case mémoire m puis à évaluer e_2 en accédant à la case mémoire m à chaque utilisation de x.

On cherche à produire, pour chaque expression E, du code de la machine à pile introduite en cours dont l'exécution a pour effet de calculer la valeur de l'expression E au sommet de la pile. On suppose dans la suite que dans l'expression E, toutes les variables locales ont un nom différent. On choisit de représenter chaque variable locale à un emplacement fixé à partir de la base de la pile.

1. On suppose donnée une fonction `index` qui à chaque variable locale x de E, associe un entier n qui représente le décalage par rapport à la base de la pile à laquelle la variable x est stockée. On suppose que le nombre de cases mémoire différentes à utiliser pour stocker les variables locales de l'expression E est N. On suppose également que pour chaque fonction prédéfinie f, il y a un label `label`(f) auquel sont associées les instructions machines correspondant au calcul de f. Ces fonctions prennent en argument deux entiers et renvoient un entier, cet entier est stocké à la place du premier argument.

   Proposer une fonction `code` qui produit pour chaque expression E, les instructions de la machine à pile nécéssaires à l'évaluation de cette expression.
2. Soit l'expression

   ```
   let x = 1 in
   let t = let u = 2 in plus(u,3) in
   f(let y = h(2,t) in g(y,x), let z=4 in g(z,z))
   ```

On suppose que x est stocké au décalage 0, t et u en 1 et y et z en 2. Donner le code produit par votre programme pour cette expression.

3. On se propose maintenant de calculer une fonction `index` correcte. On adopte la stratégie suivante : si en un point de programme, les variables visibles sont $x_0, x_1, \ldots, x_n$ alors chaque x_i aura pour décalage i, c'est-à-dire que l'on aura $\texttt{index}(x_i) = i$.
   Proposer une fonction `analyse` qui étant donnée une expression E, attribue à chaque variable locale de E un décalage. On pourra supposer donnée une fonction `set_index` telle que (`set_index` n x) remplit une table utilisée par la fonction `index` en associant le décalage n à la variable x.
4. Proposer une fonction `maxvar` qui calcule pour chaque expression E le nombre de cases mémoire à réserver pour les variables locales en utilisant la stratégie précédente.
   Combien de cases mémoire doit-on utiliser pour l'expression de la question 2 ?.
5. Proposer un décalage pour les variables de l'expression de la question 2 qui n'utilise que deux cases mémoire en supposant que dans l'expression $f(e_1, e_2)$, l'expression e_1 est évaluée avant e_2.
6. On se propose de mettre en œuvre une analyse plus fine pour le calcul de `index` qui prenne en compte l'utilisation effective des variables. Pour réaliser cette analyse, on transforme l'expression en une suite d'affectations élémentaires de la forme $x := t$ avec x une variable, et t un terme simple qui peut être une constante c, une variable y ou bien $f(y, z)$ l'application d'une fonction f à deux variables y et z. La variable x dans laquelle le résultat de t est stocké peut être soit une variable du programme, soit désigné le sommet de pile.
   On introduit le langage intermédiaire suivant:

```
type v =  Var of string | Stack of string

type sexpr =
    Scte of int
  | Svar of v
  | Sfun of string * v * v
```

   Pour compiler une expression e on se donne une variable r dans laquelle on souhaite stocker la valeur de e. Si e est une constante ou une variable, on utilise l'affectation correspondante. Si e est de la forme `let` x $= e_1$ `in` e_2 on engendre les affectations élémentaires pour calculer la valeur de l'expression e_1 dans la variable x puis les affectations pour calculer e_2 dans la variable r. Si e est de la forme $f(e_1, e_2)$, on crée des nouvelles variables r_1 et r_2, on engendre les affectations pour calculer e_1 dans r_1 puis e_2 dans r_2 et on ajoute l'affectation $r := f(r_1, r_2)$, si e_1 (resp. e_2) est déjà une variable, on peut l'utiliser directement.
   (a) Donner la suite d'affectations correspondant à l'expression de la question 2.
   (b) Fournir pour chaque point du programme ainsi obtenu l'ensemble des variables vivantes. On ne prendra pas en compte les variables stockés dans la pile. Construire le graphe d'interférence et en déduire une manière d'allouer chacune des variables.
   (c) Proposer une fonction `transforme` qui étant données une variable r et une expression e, calcule la suite d'affectations équivalente pour le calcul de la valeur de e dans la variable r.
7. On se donne une suite d'affectations p, proposer une fonction `vivantes` qui calcule l'ensemble des variables vivantes au début de p (qui seront donc utilisées dans p avant d'être redéfinies). On pourra utiliser les notations ensemblistes suivante : l'ensemble vide `empty`, le singleton `(singleton x)`, l'union de deux ensembles `(union e1 e2)` et la différence entre deux ensembles `(diff e1 e2)`.

18 Filtrage

(Examen janvier 04)

On étudie un langage permettant la définition d'expressions par filtrage.

Exemple Dans un tel langage, on peut par exemple introduire un nouveau type *tree* pour représenter les arbres, avec deux constructeurs *Leaf* et *Node*.

$$\texttt{type}\ \mathit{tree}\ =\ \mathit{Leaf}\ \texttt{of int}\ |\ \mathit{Node}\ \texttt{of int} * (\mathit{tree} * \mathit{tree})$$

L'expression suivante, définie par filtrage, calcule la valeur à la racine de l'arbre e :

$$\begin{array}{lll} \texttt{match}\ e\ \texttt{with} & \mathit{Node}(x,(y,z)) & \texttt{->}\ x \\ | & \mathit{Leaf}(x) & \texttt{->}\ x \end{array}$$

On peut également former des motifs plus complexes comme dans l'expression suivante notée E_0 dans la suite de l'exercice :

$$\begin{array}{lll} E_0\ \equiv\ \texttt{match}\ e\ \texttt{with} & \mathit{Node}(x,(y,\mathit{Leaf}(\_))) & \texttt{->}\ \texttt{plus}(x,1) \\ | & \mathit{Node}(1,(y,z)) & \texttt{->}\ 0 \\ | & \mathit{Leaf}(x) & \texttt{->}\ x \\ | & \_ & \texttt{->}\ 3 \end{array}$$

Grammaire Ce langage est décrit par la grammaire suivante dans laquelle `cte` est un token correspondant à une valeur entière, `ident` est un token correspondant à un identificateur représentant soit une variable, soit une fonction prédéfinie, soit un constructeur et finalement `let`, `in`, `match`, `with` sont des mots clés et `=`, `(`, `,`, `->`, `|`, `_` et `)` sont des symboles. E, P et B sont des non-terminaux respectivement pour les expressions, les motifs et les branches de filtrage.

$$\begin{array}{lll} E & ::= & \texttt{cte}\ |\ \texttt{ident}\ |\ \texttt{ident(}E\texttt{)}\ |\ \texttt{(}E\texttt{)}\ | \\ E & ::= & \texttt{let ident =}\ E\ \texttt{in}\ E\ |\ E\texttt{,}E\ |\ \texttt{match}\ E\ \texttt{with}\ B \\ P & ::= & \texttt{cte}\ |\ \texttt{ident}\ |\ \texttt{ident(}P\texttt{)}\ |\ \texttt{\_}\ |\ P\texttt{,}P\ |\ \texttt{(}P\texttt{)} \\ B & ::= & P\texttt{->}E \\ B & ::= & P\texttt{->}E\ \texttt{|}\ B \end{array}$$

Types Les expressions de ce langage sont typées. Les types sont soit le type primitif `int`, soit un type produit $\tau_1 * \tau_2$ soit un identificateur correspondant à un type défini par l'utilisateur. Un tel type est introduit par une déclaration de type de la forme :

$$\texttt{type}\ t = C_1\ \texttt{of}\ \tau_1\ |\ \ldots\ |\ C_n\ \texttt{of}\ \tau_n$$

où $C_1, \ldots, C_n$ sont les constructeurs de t et τ_i est le type de l'argument de C_i. On suppose données deux fonctions :

- `def` associe à un identificateur de type t la liste de ses constructeurs avec le type de leur argument : $\texttt{def}(t) = [(C_1, \tau_1)\ ;\ \ldots\ ;\ (C_n, \tau_n)]$
- `profil` prend en argument un identificateur F (correspondant à une fonction ou un constructeur) et renvoie le profil de cet identificateur sous la forme d'un couple (τ, τ') avec τ le type de l'argument attendu par F et τ' le type du résultat de l'application de F à une expression e de type τ. Pour le constructeur C_i du type t on a $\texttt{profil}(C_i) = (\tau_i, t)$. Si `plus` correspond à l'addition sur les entiers, on aura $\texttt{profil}(\texttt{plus}) = (\texttt{int} * \texttt{int}, \texttt{int})$.

Environnement Un environnement associe des types à des variables. On note $[x_1 : \tau_1; \ldots; x_n : \tau_n]$ l'environnement qui associe à chaque variable x_i le type τ_i. Soit Γ cet environnement, on définit $\texttt{Var}(\Gamma)$ l'ensemble $\{x_1, \ldots, x_n\}$ des variables de l'environnement. On note $\Gamma(x_i)$ le type τ_i associé à x_i dans Γ. La concaténation de l'environnement $\Gamma' = [y_1 : \sigma_1; \ldots; y_m : \sigma_m]$ à Γ est un environemment noté $\Gamma;\Gamma'$ défini comme $[x_1 : \tau_1; \ldots; x_n : \tau_n; y_1 : \sigma_1; \ldots; y_m : \sigma_m]$.

Évaluation du filtrage Une valeur v est soit une constante c, soit une valeur construite $C(v')$ avec C un constructeur, soit la paire (v_1, v_2) de deux valeurs v_1 et v_2.
Une valeur v filtre un motif p dont les variables sont $\{x_1, \ldots, x_n\}$ s'il existe des valeurs $v_1, \ldots, v_n$ telles que l'expression `let` $x_1 = v_1$ `in...let` $x_n = v_n$ `in` p s'évalue en la valeur v. La liste des couples $[(x_1, v_1); \ldots; (x_n, v_n)]$ est la solution du filtrage. On peut réaliser une fonction de filtrage `filtre` d'une valeur v par un motif p de la manière suivante :

`filtre`(v, x)	$= [(x, v)]$	si x est une variable
`filtre`$(v, \_)$	$= []$	
`filtre`(c, c)	$= []$	si c est une constante
`filtre`$(C(v), C(p))$	$=$ `filtre`(v, p)	C est un constructeur
`filtre`$((v_1, v_2), (p_1, p_2))$	$=$ `filtre`(v_1, p_1)@`filtre`(v_2, p_2)	
`filtre`(v, p)	$=$ `erreur`	dans tous les autres cas

L'objectif de cet exercice est d'étudier différents aspects de la compilation d'un langage avec filtrage (analyse syntaxique, typage et génération de code).

1. **Analyse syntaxique** La grammaire suivante est une version simplifiée de la grammaire des motifs avec `atom` un terminal.

$$P ::= \texttt{atom} \mid P, P \mid \texttt{(}P\texttt{)}$$

   (a) La grammaire précédente est ambiguë, expliquer pourquoi.

   (b) Construire l'automate SLR(1) de cette grammaire, dans quel état a lieu le conflit, quelle est la nature de ce conflit ?

   (c) On souhaite que l'expression p_1, p_2, p_3 soit interprétée comme $p_1, (p_2, p_3)$, donner les transitions à effectuer à partir de l'état conflictuel pour obtenir ce comportement.

2. **Typage des motifs** Un motif p est bien formé si on peut lui associer un type τ et un environnement $l = [x_1 : \tau_1; \ldots; x_n : \tau_n]$ tel que
   – $\texttt{Var}(l) = \{x_1, \ldots, x_n\}$ sont les variables introduites par le motif p;
   – chaque variable x_i apparaît exactement une fois dans le motif p;
   – dans l'environnement l le motif p est bien formé de type τ.
   On note cette relation $l \vdash p : \tau$. On la définit formellement par les règles suivantes :

$$\frac{c \text{ est une constante entière}}{[] \vdash c : \texttt{int}} \qquad \frac{}{[] \vdash \_ : \tau} \qquad \frac{}{[x : \tau] \vdash x : \tau}$$

$$\frac{l \vdash p : \tau' \quad C \text{ constructeur de profil } (\tau', \tau)}{l \vdash C(p) : \tau} \qquad \frac{l_1 \vdash p_1 : \tau_1 \quad l_2 \vdash p_2 : \tau_2 \quad \texttt{Var}(l_1) \cap \texttt{Var}(l_2) = \emptyset}{l_1; l_2 \vdash p_1, p_2 : \tau_1 * \tau_2}$$

   (a) On appelle p_1, p_2, p_3, p_4 les motifs de l'expression E_0 introduite au début de l'exercice. Donner les environnements l_1, l_2, l_3, l_4 tels que $l_i \vdash p_i : \mathit{tree}$

   (b) Proposer une fonction qui étant donnés un motif p et un type t calcule l tel que $l \vdash p : t$ si le motif p est bien formé de type t et échoue sinon.

3. **Typage des expressions** Les règles de typage du langage définissent sous quelles conditions une expression e est bien typée de type τ dans un environnement Γ. On note cette relation $\Gamma \vdash e : \tau$.

$$\frac{c \text{ est une constante entière}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau' \quad \texttt{profil}(F) = (\tau', \tau)}{\Gamma \vdash F(e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : t \quad l_1 \vdash p_1 : t \quad \Gamma; l_1 \vdash e_1 : \tau \;\ldots\; l_n \vdash p_n : t \quad \Gamma; l_n \vdash e_n : \tau}{\Gamma \vdash \texttt{match } e \texttt{ with } p_1 \texttt{ -> } e_1 \mid \ldots \mid p_n \texttt{ -> } e_n \;: \tau}$$

   Proposer une fonction `typeof` qui étant donnés un environnement Γ et une expression e calcule le type τ tel que $\Gamma \vdash e : \tau$ si un tel type existe et échoue sinon.

4. **Élimination du filtrage** Le filtrage permet de faire des constructions de programme par cas (comme la conditionnelle) et d'accéder aisément aux sous-expressions d'un terme contenant des paires et des constructeurs. On se propose de transformer une expression contenant du filtrage en une expression sans filtrage mais utilisant les constructions supplémentaires suivantes :
   - des conditionnelles booléennes `if` E `then` E `else` E;
   - des fonctions `fst` et `snd` d'accès aux composantes d'une paire : si e est une expression de type $\tau_1 * \tau_2$ qui s'évalue en la paire (v_1, v_2) alors `fst`(e) (resp. `snd`(e)) est une expression de type τ_1 (resp. τ_2) qui s'évalue en la valeur v_1 (resp. v_2);
   - Une fonction de test d'égalité sur les entiers notée $e_1 = e_2$.
   - Une fonction `test` qui étant donnés une expression E et un constructeur C, renvoie vrai si l'expression E s'évalue en une valeur $C(v)$ et faux sinon;
   - Pour chaque constructeur C de profil (τ', τ), une fonction $\texttt{proj}_C$ de profil (τ, τ') telle que $\texttt{proj}_C(E)$ s'évalue en la valeur v si l'expression E s'évalue en une valeur $C(v)$ et échoue sinon.

   (a) Proposer une fonction `transforme` qui prend en argument une expression e, un motif p, une expression *succès* pouvant mentionner les variables du motif p et une expression *échec*, et qui calcule une expression ne comportant pas d'expression `match` mais pouvant contenir les fonctions et constructions données ci-dessus, dont le comportement est le même que celui de l'expression : `match` e `with` p `->` `succès` | `_ ->` `échec`.
   C'est-à-dire que si e s'évalue en une valeur v, si `filtre`(v, p) renvoie la séquence $[(x_1, v_1), \ldots, (x_n, v_n)]$ alors le résultat attendu s'évalue comme `let` $x_1 = v_1$ `in` $\ldots$ `let` $x_n = v_n$ `in` *succès* sinon le résultat attendu est *échec*.
   On construira la fonction `transforme` par cas suivant le motif p. Les cas du caractère `_` et d'une constante c peuvent se définir de la manière suivante :

   $$\texttt{transforme}(e, \texttt{\_}, \textit{succès}, \textit{échec}) = \textit{succès}$$
   $$\texttt{transforme}(e, c, \textit{succès}, \textit{échec}) = \texttt{if eq}(e, c)\ \texttt{then}\ \textit{succès}\ \texttt{else}\ \textit{échec}$$

   (b) En déduire une fonction `élimine` qui étant donnée une expression e remplace les constructions `match` par des conditionnelles simples et des fonctions de test et de projection. On supposera que dans les constructions `match` de e, les filtrages sont exhaustifs c'est-à-dire qu'il y a toujours un motif pour lequel le filtrage réussit.

   (c) Donner la traduction de l'expression `match` e `with` *Node*$(x, \texttt{\_}, \texttt{\_})$`->` x | *Leaf*(x)`->` x.

5. **Génération de code** On se propose maintenant de compiler le langage sans filtrage dans le code de la machine à pile vue en cours. Pour cela on décide de représenter les valeurs entières par des entiers, et les valeurs paires (v_1, v_2) ou constructeur $C(v)$ par des adresses dans le tas. Cette adresse pointe sur un bloc de deux cases mémoire. Dans le cas d'une paire, la première case contient la valeur v_1 et la seconde la valeur v_2 (v_1 et v_2 peuvent être une adresse ou un entier) . Dans le cas d'une valeur constructeur $C(v)$, la première case contient un entier représentant l'étiquette du constructeur C (attribuée de manière statique) et la seconde contient la valeur v. On suppose qu'une fonction `tag` fournit pour chaque constructeur C, le numéro de l'étiquette de C.

   Soit `code` la fonction qui étant donnée une expression sans filtrage produit le code pour la machine à pile permettant de calculer la valeur de cette expression. Expliciter la définition de `code` dans les cas suivants :
   Application $C(e)$ et fonction de test `test`(C, e) avec C un constructeur et e une expression; paire e_1, e_2 de deux expressions e_1 et e_2; projection `fst`(e) de l'expression e sur la première composante.

19 Exhaustivité et compilation du filtrage

Dans les langages fonctionnels comme Ocaml, le filtrage est une construction centrale, apparaissant dans les définitions de fonctions

$$\texttt{function}\ pat_1 \to e_1 \mid \ \ldots\ \mid pat_n \to e_n$$

les conditionnelles généralisées

$$\texttt{match}\ e\ \texttt{with}\ pat_1 \to e_1 \mid \ \ldots\ \mid pat_n \to e_n$$

ou encore les *handlers* d'exceptions

$$\texttt{try}\ e\ \texttt{with}\ pat_1 \to e_1 \mid \ \ldots\ \mid pat_n \to e_n$$

Le but de cet exercice est d'introduire un algorithme de *compilation du filtrage*, c'est-à-dire de transformation d'un filtrage en séquence de tests élémentaires (du type `if then else`).

Le langage des *motifs* est le suivant

$$p \ ::= \ x \mid C(p_1, \ldots, p_n)$$

où x désigne une variable et C un constructeur d'arité n. Un constructeur C peut être constant (`[]`, `false`, `true`, `0`, `1`, `"chaine"`, ...) ou d'arité $n > 0$ (`::`, constructeur de n-uplet, ...). Certains types ont un nombre fini de constructeurs (types construits, booléens, ...), d'autres une infinité de constructeurs (entiers, chaînes, ...).

Dans tout cet exercice, on se restreint aux motifs *linéaires* : une même variable apparaît au plus une fois dans un même motif. Ainsi le motif (x, x) n'est pas linéaire, mais (x, y) l'est.

L'ensemble des *valeurs* est le suivant

$$v \ ::= \ C(v_1 \ldots, v_n)$$

On dit qu'un motif p *filtre* une valeur v s'il existe une substitution σ (de variables par des valeurs) telle que $\sigma(p) = v$. Pour un filtrage a plusieurs cas de la forme

$$\texttt{match}\ e\ \texttt{with}\ pat_1 \to e_1 \mid \ \ldots\ \mid pat_n \to e_n$$

on dit que la valeur v de e est filtrée par le i-ème cas du filtrage si p_i filtre v et pour tout $j < i$, v n'est pas filtrée pas p_j. Le résultat du filtrage est alors $\sigma(e_i)$, où σ est la substitution telle que $\sigma(p_i) = v$. Si v n'est filtrée par aucun des p_i alors le filtrage produit une erreur à l'exécution (exception `Match_failure` en Ocaml) et on dit alors qu'il est *non-exhaustif*.

Notations : dans la suite, on notera *construct*(v) le constructeur d'une valeur v, et $\#(i)v$ la i-ème composante de la valeur v (le cas échéant). Autrement dit, *construct*$(C(v_1, \ldots, v_n)) = C$ et $\#(i)C(v_1, \ldots, v_n) = v_i$.

1. Montrer que l'ensemble des valeurs filtrées par $C(p_1, \ldots, p_n)$ est l'ensemble des $C(v_1, \ldots, v_n)$ où chaque p_i filtre v_i pour $i = 1, \ldots, n$.
2. Proposer une compilation simple du filtrage *à une ligne* sous la forme d'une fonction $F(p, v, e)$ où p est le motif, v la valeur filtrée et e l'action à effectuer en cas de succès (une expression).

   Appliquer cet algorithme au filtrage $\texttt{match}\ x\ \texttt{with}\ 1\ ::\ y\ ::\ z \to y + \texttt{length}(z)$.
3. Généraliser cette compilation au cas de plusieurs lignes. Indication : considérer maintenant une fonction $F(p, v, e, \textit{échec})$ où *échec* est l'action à effectuer en cas d'échec du filtrage de v par p.

   Appliquer cet algorithme au filtrage $\texttt{match}\ x\ \texttt{with}\ [] \to 1 \mid 1\ ::\ y \to 2 \mid z\ ::\ y \to z$.

   Qu'observez-vous concernant l'efficacité de cet algorithme ?

4. Pour faire mieux, il faut considérer le filtrage dans sa globalité (et non pas ligne à ligne) et considérer également le filtrage simultané de plusieurs valeurs. On représente donc le problème de filtrage par une matrice de la forme

$$\left|\begin{array}{cccc|c} v_1 & v_2 & \dots & v_m & \\ p_{1,1} & p_{1,2} & \dots & p_{1,m} & \to e_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \dots & p_{n,m} & \to e_n \end{array}\right|$$

où les v_i sont les m valeurs à filtrer, les $p_{i,j}$ les motifs, et les e_i les actions. Une telle matrice correspond donc au filtrage suivant :

$$\begin{array}{l} \texttt{match } (v_1, \dots, v_m) \texttt{ with} \\ \mid (p_{1,1}, \dots, p_{1,m}) \to e_1 \\ \mid \dots \\ \mid (p_{n,1}, \dots, p_{n,m}) \to e_n \end{array}$$

L'algorithme de compilation du filtrage se présente comme une fonction F prenant une telle matrice en argument et définie ainsi :

– si $n = 0$ alors

$$F(|v_1 \dots v_m|) = \mathit{erreur}$$

– si $m = 0$ alors

$$F\left(\left|\begin{array}{c} \to e_1 \\ \vdots \\ \to e_n \end{array}\right|\right) = e_1$$

– si toute la colonne de gauche se compose de variables, i.e.

$$M = \left|\begin{array}{cccc|c} v_1 & v_2 & \dots & v_m & \\ x_{1,1} & p_{1,2} & \dots & p_{1,m} & \to e_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n,1} & p_{n,2} & \dots & p_{n,m} & \to e_n \end{array}\right|$$

alors on peut supprimer cette première colonne :

$$F(M) = F\left(\left|\begin{array}{ccc|c} v_2 & \dots & v_m & \\ p_{1,2} & \dots & p_{1,m} & \to \texttt{let } x_{1,1}{=}v_1 \texttt{ in } e_1 \\ \vdots & \ddots & \vdots & \vdots \\ p_{n,2} & \dots & p_{n,m} & \to \texttt{let } x_{n,1}{=}v_1 \texttt{ in } e_n \end{array}\right|\right)$$

– enfin, si la colonne de gauche contient au moins un motif n'étant pas une variable, on construit pour chaque constructeur C apparaissant en tête d'un motif de la première colonne une sous-matrice MC regroupant les lignes commençant par le constructeur C ainsi que les lignes commençant par une variable. Dans cette sous-matrice MC, la première colonne est remplacée par k colonnes correspondant aux k arguments du constructeur C. Les actions sont inchangées, sauf dans le cas où le motif était une variable x et où l'action e devient alors `let` $x{=}v_1$ `in` e.
On construit également la sous-matrice MR regroupant les lignes commençant par une variable, où la première colonne est supprimée et les actions modifiées comme ci-dessus. Si $C_1, \dots, C_p$ sont les p constructeurs apparaissant en tête dans la première colonne de M, on définit alors

$$\begin{array}{rl} F(M) = & \texttt{if } \mathit{construct}(v_1) = C_1 \texttt{ then } F(MC_1) \texttt{ else} \\ & \vdots \\ & \texttt{if } \mathit{construct}(v_p) = C_p \texttt{ then } F(MC_p) \texttt{ else} \\ & F(MR) \end{array}$$

Pour compiler le filtrage `match` v `with` $p_1 \to e_1$ `|` ... `|` $p_n \to e_n$, il suffit alors de lancer F sur la matrice

$$M = \left| \begin{array}{ll} v & \\ p_1 & \to e_1 \\ \vdots & \\ p_n & \to e_n \end{array} \right|$$

(a) Appliquer cet algorithme sur les exemples suivants :

$$\texttt{match } v \texttt{ with [] } \to 1 \mid 1 :: y \to 2 \mid z :: y \to z$$

```
match x with
| (true, true) → true
| (y, false) → false
| (false, y) → false
```

```
match v with
| C(q) → e1
| D → e2
| x → e3
| E(r, s) → e4
| y → e5
| C(t) → e6
| E(u, v) → e7
```

(b) Que peut-on dire du nombre de tests effectués ? de la taille du code produit ?

(c) En quoi cet algorithme permet-il de décider le problème de l'exhaustivité du filtrage (i.e. décider si toute valeur possible est bien couverte par au moins un motif) ?

5. Montrer que le problème de l'exhaustivité du filtrage est NP-dur. Indication : chercher à réduire le problème SAT (satisfiabilité d'une formule booléenne) au problème de l'exhaustivité.

20 Compilation d'un petit langage fonctionnel

Les langages fonctionnels permettent d'écrire des fonctions imbriquées et de prendre en argument ou de rendre en résultat des fonctions. Dans ces langages, les fonctions sont des "citoyens de première classe", c'est à dire des valeurs comme les autres.

On se retrouve ici confronté à un problème sémantique lié à la présence de "variables libres", c'est-à-dire de variables n'apparaissant ni comme paramètres, ni comme variables globales. Ainsi, la fonction `add` ci-dessous contient une fonction qui accède à la variable libre `x`.

```
let add = fun x -> (fun y -> x + y + 2)
let add1 = add 42
let r = add1 0
```

Où doit-on stocker `x` pour que la fonction imbriquée `fun y -> x + y + 2` puisse accéder à sa valeur? Si `x` est stocké sur la pile alors la fonction partielle `add1` sera compilée de manière incorrecte: après exécution de l'application `add 42`, la pile ne contient plus `x`. Pourtant, il paraît raisonnable de pouvoir ensuite écrire `add1 0` (qui calcule la valeur `44`).

La solution la plus simple pour implanter les fonctions de première classe est d'utiliser des *fermetures*. Une fonction `fun` $x \to e$ peut être représentée à l'exécution par une fermeture $\langle x \rightsquigarrow e, [v_1/x_1, ..., v_n/x_n] \rangle$, c'est-à-dire une paire formée du code de la fonction et de son environnement.

L'environnement contient la liste des valeurs de ses variables libres. Dans l'exemple ci-dessus, la fonction `fun y -> x + y + 2` fait référence à la variable libre `x` et la fonction `add` ne contient aucune variable libre.

L'objectif de ce TD est de proposer une méthode de compilation d'un (petit) langage fonctionnel. Cette compilation sera réalisée en deux étapes principales. La première étape est une étape d'identification des fermetures et de désimbrication. Elle permet d'obtenir un programme ne contenant plus de fonctions imbriquées. La seconde étape consiste à traduire le code intermédiaire vers du bytecode.

Avec l'exemple précédent, la première étape traduira le programme dans un programme de la forme:

```
fundef f1 [x] y -> x + y + 2
fundef f2 [] x -> clos(f1,[x])
def add = clos(f2, [])
def add1 = add 42
def r = add1 0
```

Le langage fonctionnel considéré est défini par la grammaire suivante. On supposera que les programmes sont bien typés et que tous les noms introduits sont distincts deux à deux.

```
(* syntaxe abstraite *)

type def =  Edef of (string * expr)

and expr =
   Eint of int
 | Ebool of bool
 | Elocal of string
 | Eglobal of string
 | Eop of op * expr * expr
 | Eapp of expr * expr
 | Eifthenelse of expr * expr * expr
 | Efun of string * expr
 | Elet of string * expr * expr
 | Erec of string * string * expr

and op =
    Eegal       (* =  *)
  | Enegal      (* != *)
  | Einf        (* <  *)
  | Esup        (* >  *)
  | Einf_egal   (* <= *)
  | Esup_egal   (* >= *)
  | Eplus       (* +  *)
  | Emoins      (* -  *)
  | Emult       (* *  *)
  | Ediv        (* /  *)
```

1. Ecrire une fonction `vars: expr -> string list` telle que `[vars e]` calcule la liste des variables libres de `[e]`.

   On suppose que la portée des variables a été résolue précédemment (autrement dit, toute variable est définie une seule fois)

   On définit maintenant un langage intermédiaire dans lequel il n'y a plus d'imbrications de fonctions.

```
(* syntaxe abstraite du langage sans imbrication de fonctions *)

type def =
    Mfundef of string * env * string * expr
  | Mdef of string * expr

and expr =
   Mint of int
 | Mbool of bool
 | Mlocal of string
 | Mglobal of string
 | Mop of Syntaxe_abstraite.op * expr * expr
 | Mcall of expr * expr
 | Mifthenelse of expr * expr * expr
 | Mlet of string * expr * expr
 | Mclos of string * env
 (* on ajoute les fonctions d'acces pour recuperer *)
 (* le code et l'environnement d'une fermeture *)
 | Mget_code of expr
 | Mget_env of expr

and env = string list
```

2. Donner le résultat de la traduction des déclarations précédentes.
3. Ecrire une fonction `clos: Syntaxe_abstraite.def list -> Cmm.def list` qui traduit une liste de déclarations.

Représentation des fermetures

Une fermeture est habituellement une paire composée d'une fonction et d'un environnement. La fonction prend donc un argument supplémentaire permettant d'accéder à ses variables libres.

Pour obtenir un code efficace, une fermeture est représentée ici par un tableau alloué dans le tas avec les caractéristiques suivantes. Si l'environnement contient les variables libres $x_1, ..., x_n$ alors ce tableau est de taille $n+1$ et contient les valeurs $[@f, v_1, ..., v_n]$. $@f$ désigne l'adresse de la fonction et se trouve à l'indice 0 du tableau, suivi des n valeurs des variables libres.

On définit un environnement de liaison de la manière suivante:

```
(* environnement des variables locales *)
let lenv = Hashtbl.create 1
(* environnement des variables globales *)
let genv = Hashtbl.create 1
(* environnement des variables de fermeture *)
let cenv = Hashtbl.create 1
```

Un environnement associe un déplacement entier à un identificateur.

On suppose que les instructions de la machine virtuelle sont définies dans un module `Instr`.

1. Donner le principe de la compilation d'une application de fonction `Mcall`(e_1, e_2) et la séquence d'instructions associée.

   Le tableau d'activation sera tel que le résultat de l'application est rangé à l'adresse $fp-3$, la valeur de e_2 à l'adresse $fp-2$ et la valeur de e_1 a l'adresse $fp-1$.
2. Donner le principe de la compilation d'une variable locale suivant qu'elle est définie localement ou qu'elle apparaît dans l'environnement de fermeture.

En déduire la fonction `compile_var: string -> Instr.t list` de compilation d'une variable locale.

3. Donner le principe de la compilation d'une fermeture `Mclos`(s, env) où s est un identificateur global de fonction et *env* est une liste de variables.

   En déduire une fonction `compile_clos: string -> env -> Instr.t list`.

4. Définir une fonction `compile: Cmm.expr -> Intr.t list` telle que `[compile e]` est le résultat de la compilation de l'expression `[e]`.

5. Définir une fonction `compile_decl: Cmm.def list -> Instr.t list` de compilation d'une suite de déclarations.

21 Rappels sur la machine décrite dans les notes de cours

21.1 Description

Le code intermédiaire est exécuté par une machine à pile. La pile d'exécution contient des entiers, des réels et des adresses. Les chaînes de caractères sont stockées dans une mémoire séparée et repérées par leurs adresses. Trois registres permettent d'accéder à différentes parties de la pile.

- Le registre *sp* (stack pointer) repère le sommet courant de la pile, il pointe sur la première case libre de la pile.
- Le registre *fp* (frame pointer) repère l'adresse de base des variables locales.
- Le registre *gp* contient l'adresse de base des variables globales.

La machine comporte un registre *pc* qui pointe sur l'instruction courante du code à exécuter.

Une pile annexe permet de sauvegarder les appels, elle contient des couples de pointeurs, l'un sauvegardant le registre d'instructions *pc* et l'autre le registre *fp* des variables locales.

21.2 Instructions

Les instructions sont désignées par un nom et peuvent prendre un ou deux arguments. Ceux-ci peuvent être:

- des constantes entières,
- des constantes réelles,
- des chaînes de caractères entre double quote avec les mêmes conventions que le langage C pour ce qui concerne les caractères spéciaux `\"`, `\n`, `\\`,
- des adresses référençant des emplacements dans la zone des chaînes, dans la pile principale ou dans la pile d'appels,
- une étiquette désignant un emplacement dans les instructions soit sous forme d'un entier (on référence alors la n-ème instruction à partir du début du programme) soit une étiquette symbolique.

21.2.1 Terminologie

On utilisera les conventions suivantes:

- Empiler une valeur x signifie stocker x à l'emplacement $\mathsf{P}[sp]$ et incrémenter *sp* de 1.
- Empiler n fois une valeur x signifie itérer n fois l'opération précédente.
- Dépiler n valeurs revient à décrémenter de n la valeur de *sp*.
- Le sommet de la pile représente la dernière valeur stockée dans la pile soit $\mathsf{P}[sp-1]$, le sous-somment de la pile représente l'avant dernière valeur stockée dans la pile soit $\mathsf{P}[sp-2]$.

21.2.2 Opérations de base

Les opérations arithmétiques ou flottantes s'effectuent entre le sommet et le sous-sommet de la pile, les deux arguments sont dépilés et le résultat est empilé à la place. Le résultat des

opérations de comparaison est un entier qui vaut 0 ou 1. L'entier 0 représente la valeur booléenne "faux" tandis que 1 représente la valeur booléenne "vrai".

Opérations sur les entiers

Instruction	Description
ADD	dépile n puis m qui doivent être entiers et empile le résultat $m+n$
SUB	dépile n puis m qui doivent être entiers et empile le résultat $m-n$
MUL	dépile n puis m qui doivent être entiers et empile le résultat $m \times n$
DIV	dépile n puis m qui doivent être entiers et empile le résultat m/n
NOT	dépile n qui doit être un entier et empile le résultat de $n = 0$
INF	dépile n puis m qui doivent être entiers et empile le résultat $m < n$
INFEQ	dépile n puis m qui doivent être entiers et empile le résultat $m \leq n$
SUP	dépile n puis m qui doivent être entiers et empile le résultat $m > n$
SUPEQ	dépile n puis m qui doivent être entiers et empile le résultat $m \geq n$

Opérations sur les réels

Instruction	Description
FADD	dépile n puis m qui doivent être réels et empile le résultat $m+n$
FSUB	dépile n puis m qui doivent être réels et empile le résultat $m-n$
FMUL	dépile n puis m qui doivent être réels et empile le résultat $m \times n$
FDIV	dépile n puis m qui doivent être réels et empile le résultat m/n
FINF	dépile n puis m qui doivent être réels et empile le résultat $m < n$
FINFEQ	dépile n puis m qui doivent être réels et empile le résultat $m \leq n$
FSUP	dépile n puis m qui doivent être réels et empile le résultat $m > n$
FSUPEQ	dépile n puis m qui doivent être réels et empile le résultat $m \geq n$

Opérations sur les chaînes

Instruction	Description
CONCAT	dépile n puis m qui doivent être des adresses de chaîne, empile l'adresse d'une chaîne égale à la concaténation de la chaîne d'adresse n et de celle d'adresse m.

Égalité

Le test d'égalité teste que les deux objets (entiers, réels ou adresses) sur la pile sont égaux, Une erreur d'exécution a lieu si les deux objets ne sont pas de même type. Deux chaînes égales sont stockées à la même adresse, cette instruction permet donc de tester de l'égalité de deux chaînes.

Instruction	Description
EQUAL	dépile n puis m qui doivent être de même type et empile le résultat de $n = m$

Conversions

Différentes instructions permettent de convertir une chaîne de caractères en entier ou réel et réciproquement.

Instruction	Description
ATOI	dépile l'adresse d'une chaîne et empile sa conversion en nombre entier, échoue si la chaîne ne représente pas un entier.
ATOF	dépile l'adresse d'une chaîne et empile sa conversion en nombre réel, échoue si la chaîne ne représente pas un réel.
STRI	dépile un entier et empile l'adresse d'une chaîne représentant cet entier
STRF	dépile un réel et empile l'adresse d'une chaîne représentant ce réel

21.2.3 Manipuler des données

Si x désigne une adresse dans la pile alors $x[n]$ désigne une adresse située n cases au-dessus.

Empiler

Instruction	Argument	Description
PUSHI	n entier	empile n
PUSHN	n entier	empile n fois la valeur entière 0
PUSHF	n réel	empile n
PUSHS	n chaîne	stocke n dans la zone des chaînes et empile l'adresse
PUSHG	n entier	empile la valeur située en $gp[n]$
PUSHL	n entier	empile la valeur située en $fp[n]$
PUSHSP		empile la valeur du registre sp
PUSHFP		empile la valeur du registre fp
PUSHGP		empile la valeur du registre gp
LOAD	n entier	dépile une adresse a et empile la valeur dans la pile située en $a[n]$
LOADN		dépile un entier n, une adresse a et empile la valeur dans la pile située en $a[n]$
DUPN	n entier	duplique et empile les n valeurs en sommet de pile

Dépiler

Instruction	Argument	Description
POP	n entier	dépile n valeurs dans la pile
POPN		dépile un valeur entière n sur la pile puis dépile n valeurs

Stocker

Instruction	Argument	Description
STOREL	n entier	dépile une valeur et la stocke dans la pile en fp[n]
STOREG	n entier	dépile une valeur et la stocke dans la pile en gp[n]
STORE	n entier	dépile une adresse a et une valeur v, stocke v à l'adresse $a[n]$ dans la pile
STOREN		dépile un entier n, une adresse a et une valeur v, stocke v à l'adresse $a[n]$ dans la pile

Divers

Instruction	Argument(s)	Description
CHECK	n, p entiers	vérifie que le sommet de la pile est un entier i tel que $n \leq i \leq p$, sinon déclenche une erreur
SWAP		dépile n puis m et rempile n puis m

21.2.4 Entrées-Sorties

Les instructions suivantes permettent de gérer les entrées-sorties.

Instruction	Description
WRITEI	dépile un entier et l'imprime sur la sortie standard
WRITEF	dépile un réel et l'imprime sur la sortie standard
WRITES	dépile l'adresse d'une chaîne et imprime la chaîne correspondante sur la sortie standard
READ	lit une chaîne de caractères au clavier, terminée par un retour-chariot, stocke la chaîne (sans le retour-chariot) et empile l'adresse.

21.2.5 Opérations de contrôle

Après l'exécution d'une instruction le registre *pc* est habituellement incrémenté de 1. Les opérations de contrôle permettent de modifier ce comportement par défaut.

Modification du registre *pc*

Instruction	Argument	Description
JUMP	*label* étiquette	affecte au registre *pc* l'adresse dans le programme correspondant à *label* qui peut être un entier ou une valeur symbolique.
JZ	*label* étiquette	dépile une valeur, si elle est nulle affecte au registre *pc* l'adresse dans le programme correspondant à *label* sinon incrémente *pc* de 1.
PUSHA	*label* étiquette	empile l'adresse dans le code correspondant à l'étiquette *label*

Procédure

Lors de l'appel de procédure il est nécessaire de sauvegarder les registre d'instructions et de variables locales qui seront restaurées au moment du retour.

Instruction	Description
CALL	dépile une adresse de code a, sauvegarde *pc* et *fp* dans la pile des appels, affecte à *fp* la valeur courante de *sp* et à *pc* la valeur a.
RETURN	affecte à *sp* la valeur courante de *fp*, restaure de la pile des appels les valeurs de *fp* et *pc* et incrémente *pc* de 1 pour se retrouver à l'instruction suivant celle d'appel.

21.2.6 Initialisation et fin

Dans l'état initial, le registre *pc* pointe sur la première instruction du programme. La pile des appels et la pile d'exécution sont vides. Les registre *gp* et *sp* pointent sur la base de la pile d'exécution tandis que *fp* n'est pas défini. Le registre *fp* doit être initialisé par l'instruction START qui ne peut être utilisée qu'une seule fois. Les instructions suivantes servent à stopper la machine à la fin du programme ou en cas d'erreur.

Instruction	Argument	Description
START		Affecte la valeur de *sp* à *fp*
NOP		ne fait rien.
ERR	x chaîne	déclenche une erreur d'instruction avec le message x.
STOP		arrête l'exécution du programme