Traduction des langages

1h30 avec documents

Année 2021-2022

Le barème est donné à titre indicatif. Les quatre exercices sont indépendants.

1 Compréhension du cours : les fonctions (3pt)

1. Qu'est et quel est le rôle de l'enregistrement / tableau d'activation ?
2. Qui se charge de le positionner et de l'exploiter
   (a) en TAM ?
   (b) en x86-64 ?
3. Soit le code TAM

0		JUMP main
1	f	LOADL 1
2		CALL (SB) g
3		RETURN (0) 0
4	g	LOADL 2
5		SUBR IOut
6		RETURN (0) 0
7	main	LOADL 3
8		CALL (SB) f
9		HALT

Dessiner l'état de la pile **avant** que le processeur n'exécute l'instruction 5.

2 Gestion des identifiants et génération de code (11pt)

Loop Nous souhaitons ajouter au langage RAT, vu en cours, TD et TP, les boucles "**loop**" à la Rust [1].

Le mot-clé **loop** demande d'exécuter un bloc de code à l'infini ou jusqu'à ce que le programme soit arrêté manuellement, par un ctrl-c par exemple.

Exemple

```
main{
  loop { print 0; }
}
```

L'exécution de ce programme, affiche encore et encore 0 jusqu'à ce que le programme soit arrêté manuellement.

Break Il faut donc fournir un autre moyen de sortir d'une boucle en utilisant du code. Le mot-clé **break** à l'intérieur de la boucle demande au programme d'arrêter la boucle.

1. source : `https://jimskapt.github.io/rust-book-fr/ch03-05-control-flow.html`

Exemple

```
main{
  int x = 0;
  loop {
    print x;
    if (x=10) {break;} else {x = (x+1);}
  }
}
```

L'exécution de ce programme, affiche les entiers de 0 à 10.

Boucles imbriquées S'il y a des boucles imbriquées dans d'autres boucles, **break** s'applique uniquement à la boucle de plus bas niveau. Il est possible d'associer une *étiquette de boucle* à une boucle qu'il sera ensuite possible d'utiliser en association avec **break** pour préciser que ce mot-clé s'applique sur la boucle correspondant à l'étiquette plutôt qu'à la boucle la plus proche possible.

Exemple

```
main{
  int h = 0;
  heure : loop {
    int m = 0;
    loop {
      if (m=60){break;} else {}
      if (h=24}{break heure} else {}
      print [h/m];
      m = (m+1);
    }
    h = (h+1);
  }
}
```

Ce programme affiche les heures de minuit ([0/0]) à 23h59 ([23/59]).

Masquage Contrairement à Rust qui ne génére qu'un warning, nous souhaitons que le programme suivant soit rejeté (le masquage d'un identifiant de boucle dans un bloc imbriqué est interdit) :

```
main{
  int x = 0;
  l : loop {
    print x;
    if (x=10) {break;} else {x = (x+1);}
    int y = 0;
    l : loop {
      print y;
      if (y=10) {break l;} else {y = (y+1);}
    }
  }
}
```

Syntaxe du langage RAT étendu Pour traiter ces nouvelles boucles, les règles de grammaire suivantes sont ajoutées aux règles existantes :

1. $I \rightarrow loop\ BLOC$
2. $I \rightarrow id\ :\ loop\ BLOC$
3. $I \rightarrow break;$
4. $I \rightarrow break\ id;$

Questions

1. **Gestion des identifiants**
   (a) Quelles vérifications faut-il faire sur l'identifiant de la règle 2 ?
   (b) Quelles vérifications faut-il faire sur l'identifiant de la règle 4 ?
   (c) Donner le code (OCaml ou pseudo-code) de la passe de gestion d'identifiant pour faire ces vérifications.
   (d) Votre compilateur ainsi modifié accepte-t-il le programme suivant ? Si non, quelles difficultés voyez vous à l'accepter ?

```
main{
  int x = 0;
  l : loop {
    print x;
    if (x=10) {break;} else {x = (x+1);}
  }
  int y = 0;
  l : loop {
    print y;
    if (y=10) {break l;} else {y = (y+1);}
  }
}
```

   (e) Votre compilateur ainsi modifié accepte-t-il le programme suivant ? Si non, quelles difficultés voyez vous à l'accepter ?

```
main{
  int l = 1;
  if (l>3) {
    int x = 0;
    l : loop {
      print x;
      if (x=10) {break;} else {x = (x+1);}
    }
  } else {
    l = (l+1);
  }
}
```

2. **Sur un exemple**
   (a) Dessiner l'ASTPlacement (AST avant la génération de code) de l'exemple avec les heures et les minutes.
   (b) Donner le code TAM associé à l'exemple avec les heures et les minutes.
3. **Génération de code** Donner le code (OCaml ou pseudo-code) de la passe de génération de code pour prendre en compte
   (a) la règle 1.
   (b) la règle 2.
   (c) la règle 3.
   (d) la règle 4.

3 Typage (3pt)

Nous souhaitons, comme en C, permettre d'écrire la conditionnelle sous la forme d'un opérateur ternaire (uniquement pour les affectations) :

```
variable = condition ? valeur_si_vrai : valeur_si_faux ;
```

Nous souhaitons également ajouter l'opérateur ++ sur **les entiers et les rationnels**.

Pour cela la grammaire de RAT sera complétée par les règles de production suivantes :

— $I \rightarrow id = E \ ? \ E : E$;

— $E \rightarrow id++$

Exemple

```
main{
  int v = 0;
  v = (v>0) ? v++ : 0 ;
}
```

1. Donner les nouveaux **jugements de typage** à ajouter au système de type de Rat.
2. Dessiner les AST du programme précédent avant et après la passe de typage.

4 Placement mémoire (3pt)

Nous souhaitons ajouter les listes au langage RAT vu en Cours, TD et TP.

Voilà un exemple de programme que l'on souhaite pouvoir compiler :

```
int somme (list<int> l){
  if (l=[]) {
    return 0 ;
  }
  else {
    int sommeq = call somme (tl q));
    return ((hd l)+sommeq) ;
  }
}

main {
 list <int> l1 =    (10::(1::(2::(3::[]))));
 int x = 4;
 l1 = (x::l1 );
 print call somme (l1);
}
```

Les listes sont, comme en OCaml, des structures de données homogènes (tous les éléments ont le même type) et polymorphes (nous pouvons avoir des listes de rationnels, de booléens, de listes...).

Les opérations possibles sur les listes sont :

— `[]` : la liste vide

— `hd` : accès à la tête de la liste

— `tl` : accès à la queue de la liste

— `::` : opérateur de concaténation d'un élément et d'une liste

1. Les listes doivent-elles être stockées dans la pile ou le tas ? Justifier en détail votre réponse.
2. Donner les adresses des variables du programme précédent.