Tâches et types protégés (n'existe pas en C)

Tâches

Déclaration

task [type] NomDeTâche [discriminant] is
  {entry NomDentrée [(paramètres)];}
end NomDeTâche;

Les paramètres se déclarent comme ceux des procédures et fonctions. Le discriminant se déclare comme celui d'un type article. Les types tâches peuvent être réutilisés pour construire d'autres types.

Corps d'une tâche

task body NomDeTâche is
  partie déclarative
begin
  instructions de la tâche
end NomDeTâche;

Le corps de la tâche peut comporter, outre les instructions usuelles, les instructions `select`, `accept` et des appels de point d'entrée.

Déclaration d'un objet-tâche

NomVariabletache : NomDeTâche [(discriminant)];

Cela ne vaut que pour les types tâches.

Types protégés

Déclaration

Protected NomDeType [discriminant] is
  {function NomFns [(paramètres)] return NomType;}
  {procedure NomProc [(paramètres)];}
  {entry NomDentrée [(paramètres)];}
private
  éléments protégés
end NomDeTâche;

Les paramètres se déclarent comme ceux des procédures et fonctions.

Corps d'un type protégé

Protected body NomDeType is
  corps des fonctions, procédure et entrées
end NomDeType;

Le corps des fonctions et procédures se déclare comme habituellement. Pour les entrées, on doit respecter la syntaxe suivante:

entry NomDentrée [(discriminant)]
  when condition is
begin
  instructions de l'entrée
end NomDentrée;

Appel d'un point d'entrée, accept et select

Appel d'un point d'entrée

NomTâche.NomDentrée [(paramètres)];

Instruction accept

accept NomDentrée [(paramètres)] do
  instructions à exécuter
end NomDentrée;

Instruction select

select
  AlternativeDeSelect;
{or
  AlternativeDeSelect;}
[else
  Instructions
end select;

Alternatives dans une instruction select

Alternative d'appel d'un point d'entrée

une instruction d'appel de point d'entrée normale.

Alternative accept

une instruction accept normale éventuellement gardée. La syntaxe de la garde est la suivante:

when condition =>

Alternative delay

delay ValeurDecimale;

l'alternative se décenche au bout de *ValeurDecimale* secondes:

Alternative de terminaison

terminate;

se déclenche «quand il faut» pour terminer la tâche;-).

Restrictions

L'alternative delay est incompatible avec un else dans le select. Même incompatibilité entre le else et le terminate

L'alternative delay est incompatible l'alternative terminate.

L'alternative terminate est incimpatible avec des appels de points d'entrée dans un select.

NB: faute de place, les instructions `if`/`then`/`else` et `case`/`switch` sont absentes de ce document. Mais les concepts entre C et Ada (malgré des différences de syntaxe) sont extrêmement proches.

Licence d'Informatique – Université Pierre & Marie Curie

Mémento des constructions usuelles en Ada et en C

F. Kordon, M.-F. Le Roch & F. Pétrot

Année 2009/2010 - version 1.5

L'objectif de ce document est de vous permettre de vous retrouver facilement parmi les constructions usuelles du langage Ada (que vous avez étudié au premier semestre) mais aussi de proposer des équivalents avec le langage C (que vous serez aussi amené à utliser;-). Ce document n'a pas la prétention d'être un récapitulatif complet de la syntaxe des deux langages mais une fiche de référence pour se remémorer rapidement les principales constructions (dont vous devez connaître la signification).

Ada	C
Déclaration de variable	
nomDeVariable : type [:= valeurInitiale] ;	type nomDeVariable [= valeurInitiale];
Les types de base	
integer, natural, float/digits, delta/range, character, boolean	int, unsigned, float, char, unsigned char, double, short, long
Déclaration d'un type énuméré	
type nomDeType is (listeDeValeurs) ;	typedef enum {val_1,val_2,val_3} nomDuType;
Déclaration d'un type tableau	
type NomDeType is array(IntervallesDIndices) of typeElements ;	nomDeType undeDimension[n] ; nomDeType DeuxDimensions[n][m]
Déclaration d'un type article	

type nomDeType is record
  $NomDeChamp_1$: $type_1$:= valeurInitiale ;
  $NomDeChamp_2$: $type_2$:= valeurInitiale ;
end record;
	typedef struct {
	  $type_1$ nom_1 ;
	  $type_2$ nom_2 ;
	  } nomDeType;

Déclaration d'un type article à discriminant	

type NomDeType (Discriminant) is record
  $NomDeChamp_1$: $type_1$:= valeurInitiale ;
  $NomDeChamp_2$: $type_2$:= valeurInitiale ;
end record ;
La définition des champs peut s'appuyer sur le discriminant dans une spécification de contrainte.
	Le mécanisme n'existe pas. On utilise l'allocation dynamique au moyen de pointeurs.

Déclaration d'un type accès (pointeur)	
type nomDeType is access typeElementsAccédés;	typedef typeElementAccede *NomDeType ;
Affectation	
nomDeVariable := expression ;	NomDeVariable = expression ;
Instruction bloc	

declare
  déclarations ;
begin
  instructions ;
exception
  traiteExceptions ;
end ;
Un bloc peut porter un nom suivi du caractère «:» (en amont du declare), le end du bloc est alors suivi de ce nom.
	{ déclarations ;
	  instructions ;
	}
	Les déclarations sont toujours en tête du bloc.

Sous-programmes

Ada

Les parties «déclaration» et «exception» sont optionelles. Les paramètres indiquent s'ils sont en entrée (in), en sortie (out) ou transmis par référence (in out). Un sous-programe peut être surchargé à condition d'avoir des profils de paramètres différents. Il est possible de surcharger des opérateurs prédéfinis du langage.

point d'entrée d'un programme

```
procedure nomDeProcedure is
  déclarations ;
begin
  instructions ;
exception
  traiteExceptions ;
end nomDeProcedure ;
```

procédures

```
procedure nomDeProcedure is
  déclarations ;
begin
  instructions ;
exception
  traiteExceptions ;
end nomDeProcedure ;
```

fonctions

```
function nomDeFonction(listeDeParametres) return nomDeType is
  déclarations ;
begin
  instructions contenant un «return valeur» ;
exception
  traiteExceptions ;
end nomDeFonction ;
```

C

En C, la notion de «sous-programme» n'existe pas: seules les fonctions sont connues et rendent une valeur de retour dans un type donné, (utilisation du type «void» si la fonction ne retourne rien). Les paramètres sont tous transmis par valeur. Lorsqu'un effet de bord est nécessaire, il faut transmettre l'adresse. Les déclarations sont en tête de bloc. Contrairement à Ada, il est impossible d'imbriquer des fonctions/procédure dans une fonction/procédure.

point d'entrée d'un programme

```
int main ()
int main (int argc, char* argv[])
```

procédures

```
void nomDeProcedure (){
  instructions
}
void nomDeProcedure (ListeDeParamètres){
  instructions
}
```

fonctions

```
type nomDeFonction () {
  instructions contenant un «return valeur»
}
type nomDeFonction (ListeDeParamètres) {
  instructions contenant un «return valeur»
}
```

L'ordre d'évaluation des paramètres d'une fonction n'est pas precise par la norme. Cela signifie qu'il est extrêmement dangereux, lors d'un appel de fonction, de passer en paramètre des expressions ayant un effet de bord (typiquement, des choses comme v++).

Paramètres formels de sous-programmes

Ada

```
NomFormel : mode NomDeType := ValeurInitiale
```

La valeur initiale est optionnelle et n'a de sens que pour les paramètres en mode *in*. Les modes possibles sont *in* (paramètre en lecture seulement dans le corps du sous-programme), *out* (paramètre en écriture seulement dans le corps du sous-programme) ou *in out* (paramètre en lecture et en écriture dans le corps du sous-programme).

Les fonctions ne peuvent avoir que des paramètres en mode *in*.

C

```
nomDeType nomFormel
ou
const nomDeType nomFormel
```

Pas de valeur initiale par défaut, pas réellement de mode ; «const» signifie que le paramètre ne sera pas modifié. Cette sorte de mode in n'a de sens que pour les arguments de type pointeur.

Appels de sous-programmes

Ada

Passage de paramètres avec association par ordre de déclaration des paramètres formels

ou

Passage de paramètres avec association par nom. Les paramètres associés aux paramètres formels à valeur initiale peuvent être omis.Toute valeur retournée par une fonction doit être prise en compte dans une expression.

C

Passage de paramètres avec association par ordre de déclaration des paramètres formels

Déclarations de sous-programmes

Ada

```
procedure NomDeProcedure (listeDeParamètres) ;
ou
function nomDeFonction (listeDeParamètres) return NomDeType ;
```

C

```
nomDeType NomDeFonction (liste de paramètres) ;
```

si elle est de type «void», la fonction ne rend pas de valeur (c'est une procédure).

Boucles

Ada

Les boucles peuvent être nommées (le nom est suivit du caractère «:» en amont de la boucle), le «end loop» est alors suivi du nom de la boucle. Une boucle peut contenir une instruction exit, éventuellement suivie du nom de la boucle dont on sort. Si l'instruction exit ne contient pas de nom de boucle, alors on sort de la boucle dans laquelle on se trouve.

boucle «de base»

```
loop
  instructions ;
end loop ;
```

Boucle itérative

Il n'existe pas de modèle de boucle avec test en fin de boucle en Ada. Mais avec une boucle de base et un «exit» conditionnel en dernière instruction de la boucle, on obtient le même résultat.

```
while expressionBooléenne loop
  instructions ;
end loop ;
```

Boucle énumérative

La variable VarIndice est strictement locale au corps de la boucle. Elle n'est accessible qu'en lecture.

```
for VarIndice in borneInf .. borneSup loop
  instructions ;
end loop ;
```

pour une énumération dans l'ordre inverse

```
for VarIndice in reverse borneInf .. borneSup loop
  instructions ;
end loop ;
```

Sortie d'une boucle

```
exit ;
exit nomBoucle;
ou
exit when condition;
exit nom Boucle when condition;
```

Ada ne propose pas la notion de continuation.

C

La notion de boucle est beaucoup moins structurée en C. On trouve une forme itérative mais pas réellement de forme énumérative (trop de degrés de libertés sot laissés). on peut sortir prematurement de la boucle courante avec l'instruction «break».

Boucle itérative

Deux types de boucles. La première avec test en début de boucle, la seconde avec test en fin de boucle.

```
while (condition) {
  instructions
}
ou
do {
  instructions
}
while (condition)
```

Boucle «for»

Si $expr_1$ vaut «variable = borneInf», $expr_2$ vaut «variable <= borneSup» et $expr_3$ vaut «variable++», alors on a une sémantique du type de celle de la boucle énumérative en Ada.

```
for (expr1 ; expr2 ; expr3) {
  instructions
}
```

Sortie d'une boucle

```
break ;
```

provoque une sortie de la boucle dans laquelle on se trouve.

```
continue:
```

provoque un passage à l'itération suivante de la boucle (les instructions restantes du corps sont ignorées).

Manipulation de variables dynamiques (utilisation des pointeurs)

Ada

Allocation

```
NomDeVariableAcces := new nomDeType ;
```

Ou

```
NomDeVariableAcces := new nomDeType'(valeur) ;
```

Dans le second cas, on affecte «valeur» (un agrégat si la valeur est composée) à l'espace mémoire alouée par l'instruction «new».

Accès

```
NomDeVariableAcces.all
```

correspond à la variable complète.

```
NomDeVariableAcces.nomDeChamp
```

correspond à la un champ de la variable (type article).

Désallocation

La désallocation se fait au moyen d'une procédure générique qu'il faut instancier: Unchecked_Deallocation. Cette procédure générique possède deux paramètres formels génériques (le type pointeur et le type pointé). L'instanciation possède un paramètre formel (le pointeur à désallouer). Cette procédure permet de respecter la philosophie de fort typage imposé par le langage.

C

Allocation

```
var = malloc(taille du type pointé)
```

La taille (en octets) du type pointé est obtenu par la directive de compilation «sizeof(nomType)».

L'affectation d'une valeur ne peut se faire dans la même instruction comme en Ada.

Accès

```
*nomDeVariableAcces
```

correspond à la variable.

```
nomDeVariableAcces->nomDeChamp
```

permet d'accéder à un champ de la variable quand c'est un type article. Il n'y a pas d'équivalent réel du «.all» Ada mais on peut recopier les variables de types struc directement. Par contre, ce n'est pas le cas des pointeurs sur des tableaux.

Désallocation

```
Free (nomDeVariableAcces);
```

La désallocation est plus «accessible» en C qu'en Ada. On vou conseille de compiler avec les options "gcc -Wall -ansi -pedantic" pour que des vérifications soient faites sur les pointeurs.