Syntaxe Haskell - Syntaxe C

$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	$\texttt{if}\,(e_1)\,\{e_2;\}\,\texttt{else}\,\{e_3;\}$
	$e_1\,?\,e_2\ :\ e_3$
$f\ x\ y\ =\ e$	$\mathit{type}_e\ f\ (\mathit{type}_x\ x, \mathit{type}_y\ y)\ \{e;\}$
$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	$\{\mathit{type}_x\ x\ = e_1;\ e_2;\}$

Types en C

- int, char, short, long, ...
- float, double, ...
- enum
- pointeurs, tableaux, ...
- void
- struct
- union
- fonctions

Un datatype en C

```
data Type = Atom | Tup Type Type | Arrow Type Type
```

```
typedef struct type type;
struct type {
  enum { ATOM, TUP, ARROW } tag;
  union {
    char *atom;
    struct { type *t[2]; } tup;
    struct { type *t1, *t2; } arrow;
  } v;
};
```

Un datatype en C, suite

```
#define XTUP(t) ((t)->v.tup)

type *type_tup (type *t1, type *t2)
{
  type *t = malloc (sizeof (type));
  t->tag = TUP;
  XTUP (t).t[0] = t1;
  XTUP (t).t[1] = t2;
  return t;
}
```

De l'assembleur au C

registres $\Longrightarrow$ variables

adresses $\Longrightarrow$ pointeurs

+struct

+union

+programmation structurée

Programmation impérative

Même modèle que la machine: *séquence* d'opérations sur la mémoire

Les *fonctions* (qui construisent un résultat) sont remplacées par des *procédures* (qui opèrent par modification de leurs arguments).

Concepts reliés:

- séquence: suite d'opérations à effectuer dans un ordre particulier, avec des dépendences implicites
- état: ce qui peut changer avec le temps
- effet de bord: modification apportée par une opération
- identité: ce qui distingue un objet d'une copie identique
- valeur: objet sans identité

Structure mémoire

La mémoire est composée d'un graphe d'objets

Mémoire = un ensemble d'objets

Objet = une séquence de bytes

- Code: ensemble de fonctions et procédures
- Pile: ensemble ordonné d'objets créés implicitement
- Tas: ensemble non-ordonné d'objets créés explicitement

Procédures

Une *fonction* ou *procédure* est un fragment de programme clairement délimité

La *signature* ou *interface* d'une fonction est son type

Le *corps* d'une fonction est le code en soi

Une définition de fonction donne son corps, sa signature, la liste des *paramètres formels*

Un appel de fonction indique la fonction et les *paramètres actuels*

Passage de paramètres

Lien entre un paramètre formel et un paramètre actuel

- par valeur (le plus répandu)
- par nom (Algol, Haskell)
- par référence (Pascal, C++, Modula-2)
- par valeur-résultat (Ada)

Passage par valeur

Le système le plus simple:

Le paramètre formel est une nouvelle variable, initialisée avec une copie de la *valeur* du paramètre actuel

Les modifications faites sur le paramètre formel n'affectent pas le paramètre actuel

I.e. l'appelé ne peut pas avoir d'effet sur l'appelant

Passage par nom

Le paramètre actuel est une *expression* passée sans être évaluée

Le paramètre formel fait référence à l'*expression*

Chaque usage du paramètre formel cause l'évaluation de l'expression

Passage par référence

Le paramètre actuel est un emplacement mémoire

Le paramètre formel désigne le même emplacement

```
PROCEDURE inc (VAR x : integer)
BEGIN
  x = x + 1;
END
... inc (z);  inc (y[3]); ...
```

On peut parfois le simuler en passant par valeur une référence:

```
void inc (int *x)
{ *x = *x + 1; }
... inc (&z);  inc (y+3); ...
```

Passage par valeur-résultat

A mi-chemin entre passage par valeur et passage par référence

Le paramètre actuel est un emplacement mémoire

le paramètre formel est une nouvelle variable initialisée avec la valeur contenue dans l'emplacement mémoire

Lorsque la fonction termine, la valeur du paramètre formel est copiée dans l'emplacement du paramètre actuel

On peut aussi le simuler en passant par valeur une référence:

```
void inc (int *xp)
{ int x = *xp; x = x + 1; *xp = x; }
... inc (&z);  inc (y+3); ...
```

Alias

alias: deux manière différentes de nommer une même entité

Example:

```
FUNCTION foo (VAR a : INTEGER) BEGIN ... END
... foo (b); ...
```

Ou encore:

```
XAPP (e1).f = XTUP (e2).e1;
```

Ou encore:

```
let x = (snd y, y) in ...
```

Alias, suite

Une modification sur un objet affecte immédiatement tous ses alias

Peut-être très utile

Mais c'est aussi une source de problèmes difficiles

Sans alias, passage par référence $\equiv$ passage par valeur-résultat

La notion d'alias est intimement liée à celle de pointeur

```
int foo (int x[], int y[])
{ x[0] = 1;
  y[0] = 2;
  return x[0] + y[0]; }
```

Pointeurs et références

Tout problème peut être résolu en ajoutant un niveau d'*indirection*

Utilisés de manière interne par les compilateurs

Utilisés pour les structures de données plus complexes que les tableaux et les enregistrements

Taille fixe (petite) indépendante de l'objet référencé

Utilisé aussi pour éviter des copies coûteuses

$$*x \quad \simeq \quad \mathit{mem}[x]$$

Opérations sur les pointeurs

Opérations principales:

- Allocation d'un objet, renvoie un nouveau pointeur sur un espace mémoire jusqu'alors inutilisé
- Indirection, pour accéder à l'objet pointé par le pointeur
- Récupération de l'espace alloué

Autres opérations parfois supportées:

- Arithmétique: $p + n$, $p_1 - p_2$, ...
- Obtention d'un pointeur sur un objet existant: $\&x$

Arithmétique sur les pointeurs

En C, on peut additionner un pointeur et un entier

$$*(x+n) \quad \simeq \quad \mathit{mem}[x+n*k]$$

C'est utilisé pour les tableaux:

$$x[n] \equiv *(x+n) \quad \equiv *(n+x) \equiv n[x]$$

Si x pointe sur un tableau de taille 10, la valeur de $x+11$ est indéfinie

On peut aussi faire la différence entre 2 pointeurs:

$$(x+n) - x == n$$

La valeur de $x_1 - x_2$ n'est pas définie si x_1 et x_2 pointent dans des objets différents

Structures de contrôle

flux/flot de contrôle = trajet suivi par le point d'exécution

flux/flot de donnée = trajet suivi par les données

- Sauts: `goto`, `break`, `continue`, ...
- Tests: `if`, `case`, filtrage, `? :`, `&&`, `||`, ...
- Boucles: `for`, `while`, `loop`, ...
- Fonctions
- Exceptions
- Non-déterminisme

Fonctions d'ordre supérieur

Les fonctions d'ordre supérieur permettent de créer de nouvelles structures de contrôle

```
fun for a b f =
    if a > b then ()
    else (f a; for (a + 1) b f)
...
for 1 10 (fn i => print (Int.toString i))
```

Programmation structurée

L'usage abusif de `goto` mène à des programmes *spaghetti*

La *programmation structurée* impose des règles qui assurent une certaine correspondance entre la structure syntaxique d'un programme et son flot de contrôle

Structuration stricte: un point d'entrée et un point de sortie

Structuration laxiste: un point d'entrée

Le switch de C

L'énoncé *switch* de C n'est pas très structuré

```
void copier (char *src, char *dst, int n)
{ while (n > 0) { *dst++ = *src++; n--; } }

void copier (char *src, char *dst, int n)
{ switch (n & 3) {
    case 0: while (n > 0) { *dst++ = *src++;
    case 3:                 *dst++ = *src++;
    case 2:                 *dst++ = *src++;
    case 1:                 *dst++ = *src++;
                            n -= 4;
}                   }         }
```

Récursion

La récursion est une forme généralisée de boucle

```
void toto ()
{
  int c, fdl;
  fdl = TRUE;
  while ((c = getchar ()) != EOF) {
    if (c == '\n') {
      if (fdl) continue;
      fdl = TRUE;
    } else
      fdl = FALSE;
    putchar (c);
} }
```

Récursion (suite)

```
void toto_1 (int fdl)
{
  int c;
  if ((c = getchar ()) != EOF) {
    if (c == '\n') {
      if (fdl) continue;
      fdl = TRUE;
    } else
      fdl = FALSE;
    putchar (c);
    toto_1 (fdl);
} }
void toto () { toto_1 (TRUE); }
```

Récursion (continue)

```
if ((c = getchar ()) != EOF) {
  if (c == '\n' && fdl)
    toto_1 (fdl);
  else {
    if (c == '\n')
      fdl = TRUE;
    } else
      fdl = FALSE;
    putchar (c);
    toto_1 (fdl);
} }
```

Récursion (nettoyage)

```
void toto_1 (int fdl)
{
  int c = getchar ();
  if (c != EOF) {
    if (c == '\n' && fdl)
      toto_1 (TRUE);
    else {
      putchar (c);
      toto_1 (c == '\n');
} } }
void toto () { toto_1 (TRUE); }
```

Récursion (fin)

```
void toto_1 (int fdl)
{
  int c = getchar ();
  if (c != EOF) {
    if (c != '\n' || !fdl)
      putchar (c);
    toto_1 (c == '\n');
} }
void toto () { toto_1 (TRUE); }
```

Représentation des objets scalaires

Les objets tels que: pointeurs, entiers, caractères, booléens, `enum`, nombres à virgule flottante, ...

Généralement traités de manière atomique

Pas de notion d'identité

Représentés par un nombre fixe de bits

Contraintes d'alignement, soit imposées par l'architecture, soit nécessaires pour des raisons de performance

Alignement naturel = adresse est un multiple de la taille de l'objet

Représentation des tableaux

Noms: tableaux, vecteurs

Disposition contiguë en mémoire

Chaque élément du tableau a la même taille

adresse de $T[i] = \mathit{base} + (i - \mathit{min}) \times \mathit{size}$

Selon les langages, la représentation peut ou non contenir un champ supplémentaire qui indique la taille totale du tableau

Sans ce champ supplémentaire le compilateur ne peut pas vérifier les dépassements de bornes

Représentation des structures

Les noms varient: structures, enregistrements, objets, tuples, ...

Dispositions contiguë des champs en mémoire:

```
record
  a      : char;      (* adr relative = 0 *)
  b1,b2  : integer;   (* adr relative = 1 et 5 *)
  c      : real;      (* adr relative = 9 *)
end
```

En réalité, du *padding* sera ajouté pour aligner le champ `b1`, donc l'adresse relative de `b1`, `b2`, et `c` sera respectivement 4, 8, et 12

Du *padding* peut ausi être ajouté à la fin

Effets de bord en Haskell

Problème:

- Déterminer si une expression est évaluée, combien de fois, et dans quel ordre, peut être très difficile en Haskell.
- Heureusement, cela ne change rien au résultat.
- Par contre, cela rend les effets de bord inutilisables.

Solution: *main* renvoie une liste de commandes à exécuter.

L'exécution de ces commandes est faite "en dehors".

`IO` τ: type d'une commande qui renverra une valeur de type τ.

Entrées sorties en Haskell

```
getChar :: IO Char
putChar :: Char -> IO ()
```

L'opérateur `>>=` permet de composer les opérations:

```
(>>=)   :: IO a -> (a -> IO b) -> IO b
getChar >>= (\c -> putChar (succ c)) :: IO ()
```

Cela s'écrit habituellement avec le sucre syntaxique `do`:

```
do c <- getChar
   putChar (succ c)
```

Affectations en Haskell

En plus de toutes les structures de données pures, Haskell a aussi des cellules spéciales que l'on peut modifier à loisir:

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

`newIORef` v crée une nouvelle cellule muable initialisée à v

`readIORef` r lit le contenu de la cellule r

`writeIORef` r v affecte à la cellule r la valeur v

Distinguer une commande et son exécution

La partie pure du langage est séparée de la partie impure

```
a = map putChar ['a', .. 'z']
b = (a,a)
c = putChar '1'
main = head (snd b)
```

a est une liste de commandes

b est une paire de 2 listes de commandes

Le programme imprime seulement "`a`"

Monads et effets

`IO` est un *monad*

- Il y en a beaucoup d'autres (*ST*, *Maybe*, *Cont*, *Parser*, *List*, *STM*, ...)
- Le monad indique quel genre d'effet peut être présent dans la commande correspondante.

Dans les langages impératifs, il existe un concept similaire: les *systèmes d'effets*

- Chaque function est annotée avec les effets qu'elle peut avoir
- E.g. $\tau_1 \xrightarrow{\epsilon} \tau_2$ où les effets de bords sont décrits par ϵ
- Les *effets* peuvent inclure la liste des exceptions potentiellement levées, les régions utilisées, ...