

# Programmation facile et robuste en Fortran : Présentation de l'outil IRPF90

*Anthony Scemama*

LCPQ-IRSAMC, Toulouse  
scemama@irsamc.ups-tlse.fr

29 Octobre 2008

# Présentation

- 1 Introduction
- 2 Programmation par IRP
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90

# Préambule

- Un programme (ou sous-programme) est une fonction des données :  
$$\text{output} = \text{programme}(\text{input})$$
- On peut toujours représenter un programme comme un graphe sans cycle
- Les nœuds du graphe sont les variables
- Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
- On parle d'*arbre de production*

# Préambule

- Un programme (ou sous-programme) est une fonction des données :  
output = programme(input)
- On peut toujours représenter un programme comme un graphe sans cycle
- Les nœuds du graphe sont les variables
- Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
- On parle d'*arbre de production*

# Préambule

- Un programme (ou sous-programme) est une fonction des données :  
output = programme(input)
- On peut toujours représenter un programme comme un graphe sans cycle
- Les nœuds du graphe sont les variables
- Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
- On parle d'*arbre de production*

# Préambule

- Un programme (ou sous-programme) est une fonction des données :  
output = programme(input)
- On peut toujours représenter un programme comme un graphe sans cycle
- Les nœuds du graphe sont les variables
- Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
- On parle d'*arbre de production*

# Préambule

- Un programme (ou sous-programme) est une fonction des données :  
output = programme(input)
- On peut toujours représenter un programme comme un graphe sans cycle
- Les nœuds du graphe sont les variables
- Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
- On parle d'*arbre de production*

## Exemple

On cherche à calculer  $t(u(d_1, d_2), v(u(d_3, d_4), w(d_5)))$  :

$$u(x, y) = x + y + 1 \quad (1)$$

$$v(x, y) = x + y + 2 \quad (2)$$

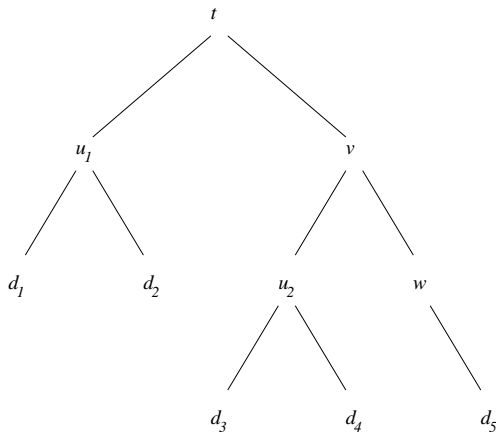
$$w(x) = x + 3 \quad (3)$$

$$t(x, y) = x + y + 4 \quad (4)$$



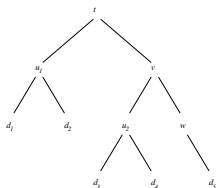
# Exemple

Arbre de production de  $t$  :



# Programmation usuelle

```
program exemple_1
  call lecture(d1,d2,d3,d4,d5)
  call calcule_u(d1,d2,u1)
  call calcule_u(d3,d4,u2)
  call calcule_w(d5,w)
  call calcule_v(u2,w,v)
  call calcule_t(u1,v,t)
  print *, t
end program
```



## Autres possibilités

```
program exemple_2
  call lecture(d1,d2,d3,d4,d5)
  u1 = calcule_u(d1,d2)
  u2 = calcule_u(d3,d4)
  w = calcule_w(d5)
  v = calcule_v(u2,w)
  t = calcule_t(u1,v)
  print *, t
end program
```

ou même :

```
program exemple3
  call lecture(d1,d2,d3,d4,d5)
  print *, t( u(d1,d2) , v( u(d3,d4),w(d5) ) )
end program
```

## Autres possibilités

```
program exemple_2
  call lecture(d1,d2,d3,d4,d5)
  u1 = calcule_u(d1,d2)
  u2 = calcule_u(d3,d4)
  w = calcule_w(d5)
  v = calcule_v(u2,w)
  t = calcule_t(u1,v)
  print *, t
end program
```

ou même :

```
program exemple3
  call lecture(d1,d2,d3,d4,d5)
  print *, t( u(d1,d2) , v( u(d3,d4),w(d5) ) )
end program
```

# Présentation

- 1 Introduction
- 2 Programmation par IRP
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90

# Programmation par IRP

```
fichier: exemple.irp.f
```

```
program exemple_4  
  print *, t  
end program
```

- L'utilisation de *t* engendre l'exploration de l'arbre.
- Équivalent à un appel de fonction mais sans paramètres (IRP : Implicit Reference to Parameters)

# Programmation par IRP

```
fichier: exemple.irp.f
```

```
program exemple_4  
  print *, t  
end program
```

- L'utilisation de *t* engendre l'exploration de l'arbre.
- Équivalent à un appel de fonction mais sans paramètres (IRP : Implicit Reference to Parameters)

# Programmation par IRP

```
fichier: exemple.irp.f
```

```
program exemple_4  
  print *, t  
end program
```

- L'utilisation de *t* engendre l'exploration de l'arbre.
- Équivalent à un appel de fonction mais sans paramètres (IRP : Implicit Reference to Parameters)



## Construction des nœuds de l'arbre

On écrit comment construire un nœud en fonction de ses fils

fichier: uvwt.irp.f

```
BEGIN_PROVIDER [ integer, t ]  
    t = u1+v+4  
END_PROVIDER
```

```
BEGIN_PROVIDER [ integer, u1 ]  
    u1 = d1+d2+1  
END_PROVIDER
```

```
BEGIN_PROVIDER [ integer, v ]  
    v = u2+w+2  
END_PROVIDER
```

## Construction des feuilles de l'arbre

On peut aussi écrire un seul fournisseur pour plusieurs variables :

fichier: input.irp.f

```
BEGIN_PROVIDER [ integer, d1 ]  
&BEGIN_PROVIDER [ integer, d2 ]  
&BEGIN_PROVIDER [ integer, d3 ]  
&BEGIN_PROVIDER [ integer, d4 ]  
&BEGIN_PROVIDER [ integer, d5 ]  
  read(*,*) d1  
  read(*,*) d2  
  read(*,*) d3  
  read(*,*) d4  
  read(*,*) d5  
END_PROVIDER
```

## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
BEGIN\_PROVIDER ... END\_PROVIDER
- Les appels call provide\_\* sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
`BEGIN_PROVIDER ... END_PROVIDER`
- Les appels `call provide_*` sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
`BEGIN_PROVIDER ... END_PROVIDER`
- Les appels `call provide_*` sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
`BEGIN_PROVIDER ... END_PROVIDER`
- Les appels `call provide_*` sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
BEGIN\_PROVIDER ... END\_PROVIDER
- Les appels call provide\_\* sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
BEGIN\_PROVIDER ... END\_PROVIDER
- Les appels call provide\_\* sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile



## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
BEGIN\_PROVIDER ... END\_PROVIDER
- Les appels call provide\_\* sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

## Génération du Fortran

- Le programme *irpf90* lit tous les fichiers \*.irp.f du répertoire courant
- Toutes les variables fournies sont répertoriées
- Toutes les variables fournies sont recherchées dans le code
- Les fournisseurs sont créés à partir des blocs  
BEGIN\_PROVIDER ... END\_PROVIDER
- Les appels call provide\_\* sont insérés dans le texte
- Chaque fichier \*.irp.f engendre un fichier \*.F90 contenant un module \*\_mod.
- Un module contient les variables fournies dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

# Présentation

- 1 Introduction
- 2 Programmation par IRP
- 3 **Fonctionnement interne de IRPF90**
- 4 Possibilités de IRPF90

## Fortran généré

```
fichier: exemple_4.F90
program exemple_4
  use uvwt_mod
  call provide_t
  print *, t
end
```

```
fichier: input.F90
module input_mod
  integer :: d1
  logical :: d1_is_built = .False.
  integer :: d5,d4,d3,d2
end module input_mod
subroutine provide_d1
  use input_mod
  if (.not.d1_is_built) then
    d1_is_built = .True.
    call bld_d1
  endif
end subroutine provide_d1
...
```

```
fichier: input.F90
...
subroutine bld_d1
  use input_mod
  read(*,*) d1
  read(*,*) d2
  read(*,*) d3
  read(*,*) d4
  read(*,*) d5
end subroutine bld_d1
...
```

```
fichier: uvwt.F90
module uvwt_mod
  integer :: u2
  logical :: u2_is_built = .False.
  integer :: u1
  logical :: u1_is_built = .False.
  integer :: v
  logical :: v_is_built = .False.
  integer :: w
  logical :: w_is_built = .False.
  integer :: t
  logical :: t_is_built = .False.
end module uvwt_mod
...
```

```
fichier: uvwt.F90
...
subroutine provide_u2
  use input_mod
  use uvwt_mod
  if (.not.u2_is_built) then
    call provide_d1
    u2_is_built = .True.
    call bld_u2
  endif
end subroutine provide_u2
...
```



```
fichier: uvwt.F90
...
subroutine bld_u2
  use input_mod
  use uvwt_mod
  integer :: fu
  u2 = fu(d3,d4)
end subroutine bld_u2
...
```

On peut utiliser des sous-routines et des fonctions Fortran, comme la fonction `fu` dans cet exemple.

# Présentation

- 1 Introduction
- 2 Programmation par IRP
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90

# Déclaration des tableaux

Par exemple, un tableau A est déclaré par :

```
BEGIN_PROVIDER [ integer, A, (dim1,dim2) ]
```

où dim1 et dim2 peuvent être :

- des entiers
- des variables fournies
- un domaine a:b entre a et b

# Déclaration des tableaux

Par exemple, un tableau A est déclaré par :

```
BEGIN_PROVIDER [ integer, A, (dim1,dim2) ]
```

où dim1 et dim2 peuvent être :

- des entiers
- des variables fournies
- un domaine a:b entre a et b

# Déclaration des tableaux

Par exemple, un tableau A est déclaré par :

```
BEGIN_PROVIDER [ integer, A, (dim1,dim2) ]
```

où dim1 et dim2 peuvent être :

- des entiers
- des variables fournies
- un domaine a:b entre a et b

Dans la routine `provide_a`, si la variable `A` n'est pas construite :

- Si le tableau `A` n'est pas encore alloué, on fait l'allocation.
- Si le tableau `A` est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des variables fournies, on appelle les fournisseurs de ces variables avant de procéder à l'allocation.

Dans la routine `provide_a`, si la variable `A` n'est pas construite :

- Si le tableau `A` n'est pas encore alloué, on fait l'allocation.
- Si le tableau `A` est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des variables fournies, on appelle les fournisseurs de ces variables avant de procéder à l'allocation.

Dans la routine `provide_a`, si la variable `A` n'est pas construite :

- Si le tableau `A` n'est pas encore alloué, on fait l'allocation.
- Si le tableau `A` est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des variables fournies, on appelle les fournisseurs de ces variables avant de procéder à l'allocation.



Dans la routine `provide_a`, si la variable `A` n'est pas construite :

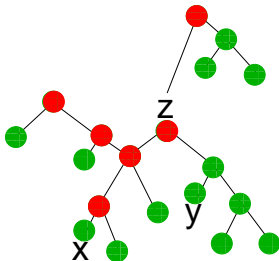
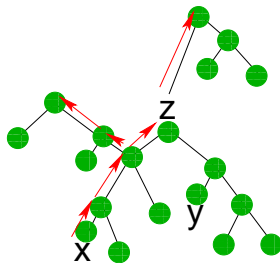
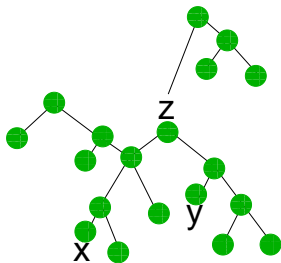
- Si le tableau `A` n'est pas encore alloué, on fait l'allocation.
- Si le tableau `A` est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des variables fournies, on appelle les fournisseurs de ces variables avant de procéder à l'allocation.

## Modification de variables fournies

La modification d'une variable fournie entraîne l'invalidation de tous ses parents. Cela est réalisé avec le mot-clé TOUCH :

```
do i=1,10
  do j=1,10
    x(j) = x(j)+3
  enddo
  TOUCH x
  y(i) = x
enddo
```

Si y et x ont un parent z en commun celui-ci sera donc recalculé, ainsi que tous les intermédiaires entre z et y



# Assertions

Une assertion est exprimée par le mot-clé ASSERT

ASSERT (d2 > d1)

- Si l'option `-d` n'est pas présente dans la ligne de commande, les assertions sont supprimées
- Si la condition n'est pas remplie, un message d'erreur est produit à l'exécution

```
bld_A: Assert failed:  
file: input.irp.f, line: 5  
( d2 > d1 )
```

```
d2 =          2
```

```
d1 =          3
```

# Assertions

Une assertion est exprimée par le mot-clé ASSERT

ASSERT (d2 > d1)

- Si l'option `-d` n'est pas présente dans la ligne de commande, les assertions sont supprimées
- Si la condition n'est pas remplie, un message d'erreur est produit à l'exécution

```
bld_A: Assert failed:  
file: input.irp.f, line: 5  
( d2 > d1 )
```

```
d2 =          2
```

```
d1 =          3
```

# Assertions

Une assertion est exprimée par le mot-clé ASSERT

ASSERT (d2 > d1)

- Si l'option `-d` n'est pas présente dans la ligne de commande, les assertions sont supprimées
- Si la condition n'est pas remplie, un message d'erreur est produit à l'exécution

```
bld_A: Assert failed:  
file: input.irp.f, line: 5  
( d2 > d1 )
```

```
d2 =          2
```

```
d1 =          3
```

## Sauvegarde de l'état du code

L'écriture et la lecture de sous-arbres est effectuée à l'aide des mots clés `IRP_READ` et `IRP_WRITE`.

```
IRP_WRITE A  
IRP_WRITE B 2  
IRP_READ B 1
```

La commande `IRP_WRITE A` écrit dans le fichier `$prog_a_$i` le contenu de la variable `A`, et procède à l'appel de `IRP_WRITE` pour tous les fils de `A` (où `$prog` est le nom du programme et `$i` est la valeur de l'entier qui suit la commande, 0 par défaut).

De même pour la commande `IRP_READ`  
Utilisation : checkpoints.

## Introduction de scripts dans le code

Il est possible d'introduire des scripts dont le résultat est du code fortran. Par exemple :

```
fichier: exemple_5.irp.f
program example_5
  BEGIN_SHELL [ /bin/bash ]
    echo print *, \'Compiled by \'whoami\' on \'date\'\'
  END_SHELL
end
```

donne :

```
fichier: exemple_5.F90
program example_5
print *, 'Compiled by scemama on Fri Sep 5 09:14:22 CEST 20
end
```



Le bloc BEGIN\_SHELL...END\_SHELL peut aussi permettre de construire des blocs BEGIN\_PROVIDER...END\_PROVIDER :

```
BEGIN_SHELL [ /usr/bin/python ]  
for i in [1,2]:  
    var="x_"+str(i)  
    print "BEGIN_PROVIDER [ integer, "+var+" ]"  
    print " "+var+" = "+str(i)  
    print "END_PROVIDER"  
END_SHELL
```

```
BEGIN_PROVIDER [ integer, x_1 ]  
    x_1 = 1  
END_PROVIDER  
BEGIN_PROVIDER [ integer, x_2 ]  
    x_2 = 2  
END_PROVIDER
```

## Compilation conditionnelle

Il est possible de définir des variables dans la ligne de commande avec l'option `-D`. Par exemple : `irpf90 -DMPI` définit la variable `MPI`.

Ces variables définissent le code fortran à générer en utilisant la construction `IRP_IF...IRP_ELSE...IRP_ENDIF`. Par exemple :

```
subroutine abort
  IRP_IF MPI
    include 'mpif.h'
    call mpi_abort(MPI_COMM_WORLD,0,ierr)
  IRP_ELSE
    stop
  IRP_ENDIF
end
```

## Parallélisation automatique

La présence dans la ligne de commande de l'option `-openmp` parallélise automatiquement les appels `call provide_*`, et construit donc les variables en parallèle via OpenMP.

Par exemple, le code généré est :

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
    call provide_a
!$OMP SECTION
    call provide_b
!$OMP SECTION
    call provide_c
!$OMP END PARALLEL SECTIONS
```

## Debug

Chaque subroutine, fonction et fournisseur de variable contient la variable chaîne de caractères `irp_here` qui contient le nom de la routine. L'ajout de

```
print *, irp_here
```

dans le code imprime le nom de la routine courante.

La présence de l'option `-d` dans la ligne de commande imprime (pour la routine `r`) :

-> `r`                    Entrée dans la routine `r`

<- `r`    `xxx` `s`    Sortie de la routine `r`

ou `xxx` est le temps CPU passé dans `r`.

# Présentation

- 1 Introduction
- 2 Programmation par IRP
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90