Réalisation d'un compilateur

2ème partie

Analyse syntaxique

```
programme -> { declarVariable }
   { declarTableau }
   { declarFonction } '.'
declarVariable -> ENTIER IDENTIF ';'
declarTableau -> TABLEAU IDENTIF
   '[' NOMBRE ']' ';'
declarFonction -> FONCTION IDENTIF
   '(' [ IDENTIF { ',' IDENTIF } ] ')'
   { declarVariable }
   instructionBloc
...
```

Analyse syntaxique

```
programme -> { declarVariable }
   { declarTableau }
   { declarFonction } '.'
declarVariable -> ENTIER IDENTIF ';'
declarTableau -> TABLEAU IDENTIF
   '[' NOMBRE ']' ';'
declarFonction -> FONCTION IDENTIF
   '(' [ IDENTIF { ',' IDENTIF } ] ')'
   { declarVariable }
   instructionBloc
...
```

un programme correct

```
entier i;
entier j;
tableau tab[100];
fonction calcul(a, b, c)
{
   i = a * b + c;
   retour u - a;
}
fonction principale()
   entier x;
{
   x = lire();
   ecrire(calcul(x, 0, 1));
} .
```

Analyse syntaxique

```
programme -> { declarVariable }
   { declarTableau }
   { declarFonction } '.'
declarVariable -> ENTIER IDENTIF ';'
declarTableau -> TABLEAU IDENTIF
   '[' NOMBRE ']' ';'
declarFonction -> FONCTION IDENTIF
...
```

analyseur

```
void programme(void) {
   while (uc == ENTIER)
      declarVariable();
   while (uc == TABLEAU)
      declarTableau();
   while (uc == FONCTION)
      declarFonction();
   if (uc != '.')
      erreur("'.' attendu");
   uc = yylex();
}
```

Analyse syntaxique

```
programme -> { declarVariable }
   { declarTableau }
   { declarFonction } '.'
declarVariable -> ENTIER IDENTIF ';'
declarTableau -> TABLEAU IDENTIF
   '[' NOMBRE ']' ';'
declarFonction -> FONCTION IDENTIF
...
```

```
void programme(void) {
   while (uc == ENTIER)
      declarVariable();
   while (uc == TABLEAU)
      declarTableau();
   while (uc == FONCTION)
      declarFonction();
   if (uc != '.')
      erreur("'.' attendu");
   uc = yylex();
}
...
int main() {
   uc = yylex();
   programme();
   printf("Syntaxe correcte");
}
```

test de l'analyseur

Rappel : la démarche

Pour obtenir un compilateur :

1° avoir un analyseur syntaxique complet et correct (testé !)

2° ajouter l'analyse sémantique : dictionnaire des identificateurs

3° ajouter la production de code

Dictionnaire d'identificateurs

Dictionnaire ou *table d'identificateurs*

- liste de couples (*identificateur*, *propriétés*)
- grandit pendant la compilation des parties déclaratives
- n'est que consulté pendant la compilation des parties exécutables

```
entier u;
fonction calcul(a, b) {
    entier w;

    w = a - b;
    a = a * a + 2 * b;
    ...
}
```

ici (on consulte et) on augmente le dictionnaire

ici on consulte (uniquement) le dictionnaire

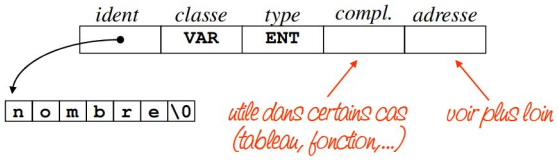
- le dictionnaire doit être réalisé avec soin : on estime qu'un compilateur passe la moitié de son temps à consulter le dictionnaire

Dictionnaire d'identificateurs

- formés de couples (*identificateur* , *propriétés*)
- dans notre cas (types simples + tableaux) :

```
entier nombre;
```

donne



Dictionnaire d'identificateurs

L'affaire des déclarations locales

- en contexte global (α) : un seul dictionnaire
- en contexte local (β) : un dictionnaire global et un local « par dessus »

```
entier x;   α

fonction truc(a, b, c)
    entier x; β
{
    x = a;
    ...
}
```

création d'un dico local

destruction du dico local

Dictionnaire d'identificateurs

L'affaire des déclarations locales

- en contexte global (α) : un seul dictionnaire
- en contexte local (β) : un dictionnaire global et un local « par dessus »

```
entier x;   α

fonction truc(a, b, c)
    entier x; β
{
    x = a;
    ...
}
```

création d'un dico local

destruction du dico local

- contrainte :

pendant la compilation d'une…	*tout identificateur doit être…*
instruction	présent dans le dico local, sinon dans le global
déclaration globale	absent du dico global
déclaration locale	absent du dico local

Dictionnaire d'identificateurs

L'affaire des déclarations locales

- en contexte global (α) : un seul dictionnaire
- en contexte local (β) : un dictionnaire global et un local « par dessus »

```
entier x;   α

fonction truc(a, b, c)
    entier x; β
{
    x = a;
    ...
}
```

création d'un dico local

destruction du dico local

- contrainte :

pendant la compilation d'une…	*tout identificateur doit être…*
instruction	présent dans le dico local, sinon dans le global
déclaration globale	absent du dico global
déclaration locale	absent du dico local

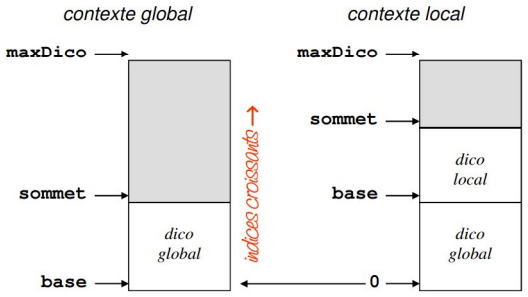
résumé :

un identificateur…	*doit être…*
…vu dans une instruction	*…présent dans au moins un dico*
…objet d'une déclaration	*…absent du dico « du dessus »*

Dictionnaire d'identificateurs

Réalisation simple et correcte :

- un tableau (**`maxDico`** est sa capacité)
- deux indices **`sommet`** et **`base`**

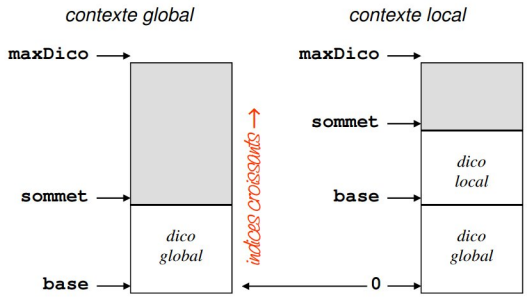


- identificateur rencontré dans une déclaration
  recherche de **`t[sommet-1]`** à **`t[base]`** (il doit être absent)
- identificateur rencontré dans une partie exécutable
  recherche de **`t[sommet-1]`** à **`t[0]`** (il doit être présent)
- tous les ajouts se font au sommet

Dictionnaire d'identificateurs

Réalisation simple et correcte :

- un tableau (**maxDico** est sa capacité)
- deux indices **sommet** et **base**



- entrée dans le contexte local (création du dico local)

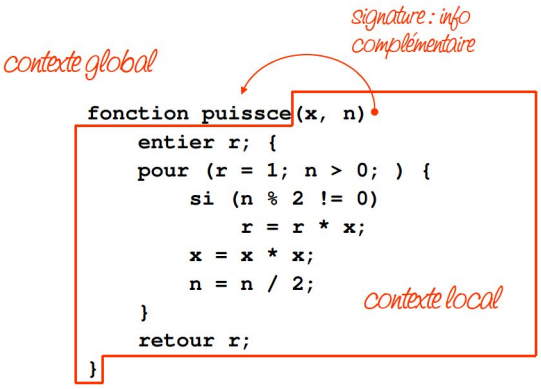
  base ← **sommet**

- sortie du contexte local (destruction du dico local)

  sommet ← **base**
  base ← **0**

Dictionnaire d'identificateurs

« entrée » et « sortie » du contexte local ?

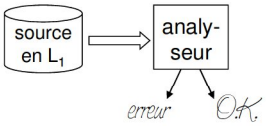


Dictionnaire d'identificateurs

- le dictionnaire est créé
  et exploité pendant la compilation
- après la compilation il n'en reste
  aucune trace

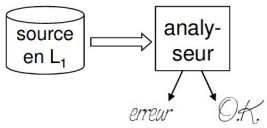
Production de code

L'*analyseur* du langage L_1

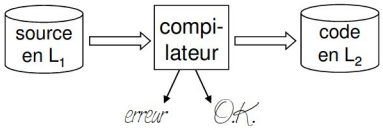


Production de code

L'*analyseur* du langage L_1



devient un *compilateur* $L_1 \rightarrow L_2$



par ajout (uniquement !) d'opérations

- de consultation/augmentation du dictionnaire
- de *production de code* machine

La machine cible

Connaissance de la « machine cible »

- son langage L_2
- la structure de sa mémoire

Ces éléments sont fixés par le fabriquant

Ici : machine cible *virtuelle* :

- *programme* qui interprète le langage L_2

La machine cible

- machine à registres

  les opérations portent sur les registres de la CPU

  ex. d'instruction : *add* R_1 R_2

La machine cible

- machine à registres

  les opérations portent sur les registres de la CPU

  ex. d'instruction : *add* R_1 R_2

- machine à pile

  les opérations portent sur le sommet d'une pile

  ex. d'instruction : *add*

La machine cible

- machine à registres
  les opérations portent sur les registres de la CPU
  ex. d'instruction : *add* R_1 R_2
- machine à pile
  les opérations portent sur le sommet d'une pile
  ex. d'instruction : *add*
- dans tous les cas, registres basiques :

  CO *compteur ordinal :*
  ex.: prochaine instruction **mem[CO]**

  SP *sommet de la pile*
  opérandes, résultats intermédiaires
  ex.: sommet de la pile **mem[SP - 1]**

  BEL *base de l'espace local*
  ex.: var. locale d'adresse i **mem[BEL + i]**

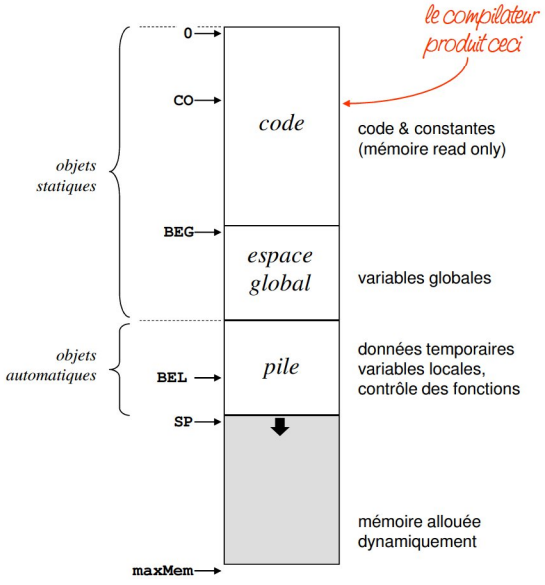
  BEG *base de l'espace global*
  ex.: var globale d'adresse i **mem[BEG + i]**

TAB. 1 – Les instructions de la machine *M*

code	*opérande*	*explication*
EMPC	valeur	*EMPiler Constante.* Empile la valeur indiquée.
EMPL	adresse	*EMPiler la valeur d'une variable Locale.* Empile la valeur de la vari déterminée par le déplacement relatif à BEL donné par adresse (entier tif).
DEPL	adresse	*DEPiler dans une variable Locale.* Dépile la valeur qui est au sommet range dans la variable déterminée par le déplacement relatif à BEL donné adresse (entier relatif).
EMPG	adresse	*EMPiler la valeur d'une variable Globale.* Empile la valeur de la vari déterminée par le déplacement (relatif à BEG) donné par adresse.
DEPG	adresse	*DEPiler dans une variable Globale.* Dépile la valeur qui est au sommet range dans la variable déterminée par le déplacement (relatif à BEG) d par adresse.
EMPT	adresse	*EMPiler la valeur d'un élément de Tableau.* Dépile la valeur qui est au son de la pile, soit i cette valeur. Empile la valeur de la cellule qui se trouve i c au-delà de la variable déterminée par le déplacement (relatif à BEG) ind par adresse.
DEPT	adresse	*DEPiler dans un élément de Tableau.* Dépile une valeur v, puis une vale Ensuite range v dans la cellule qui se trouve i cases au-delà de la vari déterminée par le déplacement (relatif à BEG) indiqué par adresse.
ADD		*ADDition.* Dépile deux valeurs et empile le résultat de leur addition.
SOUS		*SOUStraction.* Dépile deux valeurs et empile le résultat de leur soustracti
MUL		*MULtiplication.* Dépile deux valeurs et empile le résultat de leur multiplica
DIV		*DIVision.* Dépile deux valeurs et empile le quotient de leur division euclidie
MOD		*MODulo.* Dépile deux valeurs et empile le reste de leur division euclidienr
EGAL		Dépile deux valeurs et empile 1 si elles sont égales, 0 sinon.
INF		*INFerieur.* Dépile deux valeurs et empile 1 si la première est inférieure seconde, 0 sinon.
INFEG		*INFerieur ou EGal.* Dépile deux valeurs et empile 1 si la première est inféri ou égale à la seconde, 0 sinon.
NON		Dépile une valeur et empile 1 si elle est nulle, 0 sinon.
LIRE		Obtient de l'utilisateur un nombre et l'empile
ECRI		*ECRIre Valeur.* Extrait la valeur qui est au sommet de la pile et l'affiche
SAUT	adresse	*Saut inconditionnel.* L'exécution continue par l'instruction ayant l'adress diquée.
SIVRAI	adresse	*Saut conditionnel.* Dépile une valeur et si elle est non nulle, l'exécution c nue par l'instruction ayant l'adresse indiquée. Si la valeur dépilée est n l'exécution continue normalement.
SIFAUX	adresse	Comme ci-dessus, en permutant nul et non nul.
APPEL	adresse	*Appel de sous-programme.* Empile l'adresse de l'instruction suivante, puis la même chose que SAUT.
RETOUR		*Retour de sous-programme.* Dépile une valeur et continue l'exécution par l truction dont c'est l'adresse.
ENTREE		*Entrée dans un sous-programme.* Empile la valeur courante de BEL, puis c la valeur de SP dans BEL.
SORTIE		*Sortie d'un sous-programme.* Copie la valeur de BEL dans SP, puis dépile valeur et la range dans BEL.
PILE	nbreMots	*Allocation et restitution d'espace dans la pile.* Ajoute nbreMots, qui es entier positif ou négatif, à SP
STOP		La machine s'arrête.

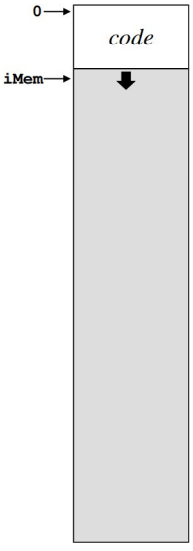
Structure de la mémoire

Mémoire de la machine cible :
dans notre cas c'est un peu plus simple



Structure de la mémoire

Mémoire de la machine cible
pendant la compilation d'un programme



Production de code

Compilation d'une expression

```
y = 123 * x + c;
```

avec

- **x** et **y** variables globales d'adresses 10 et 20
- **c** variable locale d'adresse 5

```
...
EMPC 123        empiler constante
EMPG 10         empiler var. globale
MUL             multiplier
EMPL 5          empiler var. locale
ADD             additionner
DEPG 20         depiler (dans) var. globale
...
```

Production de code

Compilation d'une expression

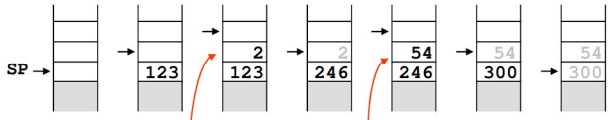
```
y = 123 * x + c;
```

avec

- **x** et **y** variables globales d'adresses 10 et 20
- **c** variable locale d'adresse 5

```
...
EMPC 123        empiler constante
EMPG 10         empiler var. globale
MUL             multiplier
EMPL 5          empiler var. locale
ADD             additionner
DEPG 20         depiler (dans) var. globale
...
```

Imaginons l'exécution :



la variable globale 10 vaut 2
la variable locale 5 vaut 54
(par exemple)

Production de code

Compilation d'une expression

```
y = 123 * x + c;
```

avec

- **x** et **y** variables globales d'adresses 10 et 20
- **c** variable locale d'adresse 5

```
...
EMPC 123
EMPG 10
MUL
EMPL 5
ADD
DEPG 20
...
```

durant la compilation on génère ceci

durant l'exécution il se passera ceci

Imaginons l'exécution :

Production de code

Que signifie « produire du code » ?

Le compilateur dépose des

- op-codes
- leurs opérandes

dans un espace (**mem**) qui peut-être

- la mémoire de la machine, ou
- destiné à être enregistré dans un fichier objet

Production de code

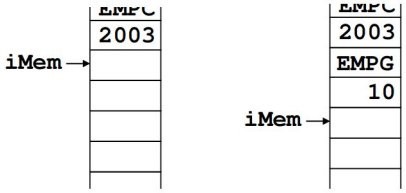
Que signifie « produire du code » ?

Le compilateur dépose des
- op-codes
- leurs opérandes

dans un espace (**mem**) qui peut-être
- la mémoire de la machine, ou
- destiné à être enregistré dans un fichier objet

Exemple: « *générer l'instruction* **EMPG 10** » :

```
mem[iMem++] = EMPG;
mem[iMem++] = 10;
```



Production de code

Extrait de la grammaire d'un langage :

…

expression → *terme* { (**'+'**|**'-'**) *terme* }

terme → *facteur* { (**'*'**|**'/'**) *facteur* }

facteur → **NOMBRE**
| **'('** *expression* **')'**
| **IDENT**
[**'['** *expression* **']'**
| **'('** [*expression* {**','** *expression*}] **')'**]

…

exemple de source correct :

123 * x + c + fon(u[i], 100) – k * (y + 1)

terme *terme* *terme* *terme*

Production de code

Compilation d'un facteur, exemple. Source :

```
... 123 ...
```

Production de code

Compilation d'un facteur, exemple. Source :

```
... 123 ...
```

grammaire concernée :

facteur → **NOMBRE** | …

Production de code

Compilation d'un facteur, exemple. Source :

```
... 123 ...
```

grammaire concernée :

facteur → **NOMBRE** | …

bout d'analyseur correspondant :

```
void facteur() {
    if (UC == NOMBRE)
        UC = uniteSuivante();
    else
        ...
}
```

Production de code

Compilation d'un facteur, exemple. Source :

```
... 123 ...
```

grammaire concernée :

facteur → **NOMBRE** | …

bout d'analyseur correspondant :

```
void facteur() {
    if (UC == NOMBRE)
        UC = uniteSuivante();
    else
        ...
}
```

compilateur :

```
void facteur() {
    if (UC == NOMBRE) {
        mem[iMem++] = EMPC;
        mem[iMem++] = atoi(lexeme);
        UC = uniteSuivante();
    } else
        ...
}
```

Production de code

Compilation d'un facteur, exemple. Source :

```
... 123 ...
```

grammaire concernée :

facteur → **NOMBRE** | …

bout d'analyseur correspondant :

```
void facteur() {
    if (UC == NOMBRE)
        UC = uniteSuivante();
    else
        ...
}
```

compilateur :

```
void facteur() {
    if (UC == NOMBRE) {
        mem[iMem++] = EMPC;
        mem[iMem++] = atoi(lexeme);
        UC = uniteSuivante();
    } else
        ...
}
```

code produit

```
...
EMPC 123
...
```

Production de code

Compilation d'un terme. Source :

```
... 123 * x ...
```

Production de code

Compilation d'un terme. Source :

```
... 123 * x ...
```

grammaire concernée :

terme → *facteur* { ('*'|'/') *facteur* }

Production de code

Compilation d'un terme. Source :

```
... 123 * x ...
```

grammaire concernée :

terme → *facteur* { ('*'|'/') *facteur* }

analyseur correspondant :

```
void terme() {
   facteur();
   while (UC == '*' || UC == '/') {
      UC = uniteSuivante();
      facteur();
   }
}
```

Production de code

Compilation d'un terme. Source :

```
... 123 * x ...
```

grammaire concernée :

terme → *facteur* { ('*'|'/') *facteur* }

analyseur correspondant :

```
void terme() {
   facteur();
   while (UC == '*' || UC == '/') {
      UC = uniteSuivante();
      facteur();
   }
}
```

compilateur :

```
void terme() {
   facteur();
   while (UC == '*' || UC == '/') {
      int ope = UC;
      UC = uniteSuivante();
      facteur();
      if (ope == '*') mem[iMem++] = MUL;
      else mem[iMem++] = DIV;
   }
}
```

Production de code

Compilation d'un terme. Source :

```
... 123 * x ...
```

grammaire concernée :

terme → *facteur* { ('*'|'/') *facteur* }

analyseur correspondant :

```
void terme() {
   facteur();
   while (UC == '*' || UC == '/') {
      UC = uniteSuivante();
      facteur();
   }
}
```

compilateur :

```
void terme() {
   facteur();
   while (UC == '*' || UC == '/') {
      int ope = UC;
      UC = uniteSuivante();
      facteur();
      if (ope == '*') mem[iMem++] = MUL;
      else mem[iMem++] = DIV;
   }
}
```

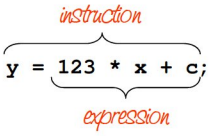
code produit

```
...
EMPC 123
EMPG 10
MUL
...
```

Production de code

À l'exécution :

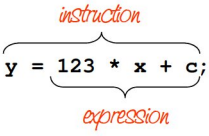
- une instruction laisse la pile comme elle l'a trouvée
- une expression ajoute une valeur au sommet

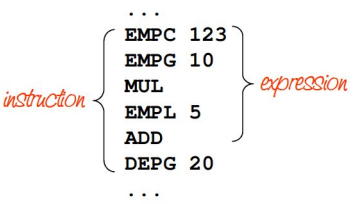


Production de code

À l'exécution :

- une instruction laisse la pile comme elle l'a trouvée
- une expression ajoute une valeur au sommet

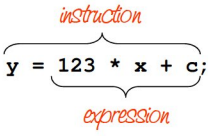


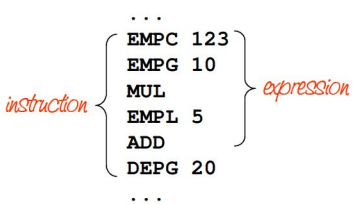


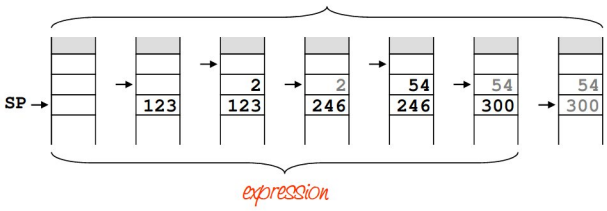
Production de code

À l'exécution :

- une instruction laisse la pile comme elle l'a trouvée
- une expression ajoute une valeur au sommet







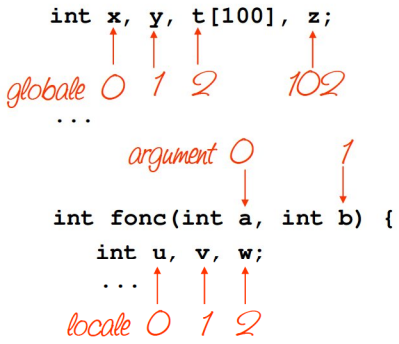
Production de code

- la question des adresses des variables, ou
- comment le compilateur alloue l'espace (non dynamique) ?

Production de code

- la question des adresses des variables, ou
- comment le compilateur alloue l'espace (non dynamique) ?

  en *comptant* l'espace utilisé par les variables rencontrées



Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

	ident	classe	type	compl.	adresse
9					
8					
7					
6					
5					
4					
3					
2					
1					
0	x	VARGLOB	ENTIER		0

Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

	ident	classe	type	compl.	adresse
9					
8					
7					
6					
5					
4					
3					
2					
1	y	VARGLOB	ENTIER		1
0	x	VARGLOB	ENTIER		0

Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

	ident	classe	type	compl.	adresse
9					
8					
7					
6					
5					
4					
3					
2	t	VARGLOB	TABLE	100	2
1	y	VARGLOB	ENTIER		1
0	x	VARGLOB	ENTIER		0

Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

	ident	classe	type	compl.	adresse
9					
8					
7					
6					
5					
4					
3	z	VARGLOB	ENTIER		102
2	t	VARGLOB	TABLE	100	2
1	y	VARGLOB	ENTIER		1
0	x	VARGLOB	ENTIER		0

Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

	ident	classe	type	compl.	adresse
9					
8					
7					
6					
base 5					
4	fonc	FONCT	ENTIER		
3	z	VARGLOB	ENTIER		102
2	t	VARGLOB	TABLE	100	2
1	y	VARGLOB	ENTIER		1
0	x	VARGLOB	ENTIER		0

Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

	ident	classe	type	compl.	adresse
9					
8					
7					
6	b	ARGUM	ENTIER		1
base 5	a	ARGUM	ENTIER		0
4	fonc	FONCT	ENTIER		
3	z	VARGLOB	ENTIER		102
2	t	VARGLOB	TABLE	100	2
1	y	VARGLOB	ENTIER		1
0	x	VARGLOB	ENTIER		0

Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

	ident	classe	type	compl.	adresse
9					
8					
7					
6	b	ARGUM	ENTIER		1
base 5	a	ARGUM	ENTIER		0
4	fonc	FONCT	ENTIER	2	
3	z	VARGLOB	ENTIER		102
2	t	VARGLOB	TABLE	100	2
1	y	VARGLOB	ENTIER		1
0	x	VARGLOB	ENTIER		0

Production de code

Allocation d'espace, adresses et dictionnaire

```
int x, y, t[100], z;
...
int fonc(int a, int b) {
    int u, v, w;
    ...
```

		ident	classe	type	compl.	adresse
	9	**w**	VARLOC	ENTIER		**2**
	8	**v**	VARLOC	ENTIER		**1**
	7	**u**	VARLOC	ENTIER		**0**
	6	**b**	ARGUM	ENTIER		**1**
base	5	**a**	ARGUM	ENTIER		**0**
	4	**fonc**	FONCT	ENTIER	**2**	
	3	**z**	VARGLOB	ENTIER		**102**
	2	**t**	VARGLOB	TABLE	100	**2**
	1	**y**	VARGLOB	ENTIER		**1**
	0	**x**	VARGLOB	ENTIER		**0**

Production de code

Séquence et rupture de séquence

CO (indice, pointeur…)
mem[CO] est *la prochaine instruction* à exécuter

Production de code

Séquence et rupture de séquence

CO (indice, pointeur…)
mem[CO] est *la prochaine instruction* à exécuter

par défaut, l'exécution est séquentielle :

CO = CO + (1 ou 2 selon l'instruction)

Production de code

Séquence et rupture de séquence

CO (indice, pointeur…)
mem[CO] est *la prochaine instruction* à exécuter

par défaut, l'exécution est séquentielle :

CO = CO + (1 ou 2 selon l'instruction)

quatre sortes de sauts (α est une adresse dans le code) :

SAUT α saut inconditionnel
CO = α

Production de code

Séquence et rupture de séquence

`CO` (indice, pointeur…)
`mem[CO]` est *la prochaine instruction* à exécuter

par défaut, l'exécution est séquentielle :

`CO = CO` + (1 ou 2 selon l'instruction)

quatre sortes de sauts (α est une adresse dans le code) :

`SAUT` α saut inconditionnel
`CO = α`

`SIFAUX` α saut conditionnel
```
if (MEM[--SP] == 0)
    CO = α
```

Production de code

Séquence et rupture de séquence

`CO` (indice, pointeur…)
`mem[CO]` est *la prochaine instruction* à exécuter

par défaut, l'exécution est séquentielle :

`CO = CO` + (1 ou 2 selon l'instruction)

quatre sortes de sauts (α est une adresse dans le code) :

`SAUT` α saut inconditionnel
`CO = α`

`SIFAUX` α saut conditionnel
```
if (MEM[--SP] == 0)
    CO = α
```

`APPEL` α appel de sous-programme
```
MEM[SP++] = CO + 2
CO = α
```

Production de code

Séquence et rupture de séquence

`CO` (indice, pointeur…)
`mem[CO]` est *la prochaine instruction* à exécuter

par défaut, l'exécution est séquentielle :

`CO = CO` + (1 ou 2 selon l'instruction)

quatre sortes de sauts (α est une adresse dans le code) :

`SAUT` α saut inconditionnel
`CO = α`

`SIFAUX` α saut conditionnel
```
if (MEM[--SP] == 0)
    CO = α
```

`APPEL` α appel de sous-programme
```
MEM[SP++] = CO + 2
CO = α
```

`RETOUR` retour de sous-programme
`CO = MEM[--SP]`

Production de code

- Exemple: compilation d'une boucle

`WHILE` *expression* **`DO`**
instruction

Production de code

- Exemple: compilation d'une boucle

WHILE *expression* **DO**
instruction

```
...
α   [ code produit par la compilation de expression (l'exécution de ce code laisse une valeur au sommet) ]
SIFAUX β
    [ code produit par la compilation de instruction ]
SAUT α
β   ...
```

Production de code

- Exemple: compilation d'une boucle

```
WHILE i < n DO
    i = i + 1;
```

Production de code

- Exemple: compilation d'une boucle

```
WHILE i < n DO
    i = i + 1;
```

```
        ...
α   100 EMPL    5
    102 EMPG   20
    104 INF
    105 SIFAUX  ?
    107
```

i est une variable locale d'adresse 5

n est une variable globale d'adresse 20

Production de code

- Exemple: compilation d'une boucle

```
WHILE i < n DO
    i = i + 1;
```

```
        ...
α   100 EMPL    5
    102 EMPG   20
    104 INF
    105 SIFAUX  ?
    107 EMPL    5
    109 EMPC    1
    111 ADD
    112 DEPL    5
    114
```

i est une variable locale d'adresse 5

n est une variable globale d'adresse 20

Production de code

- Exemple: compilation d'une boucle

```
WHILE i < n DO
    i = i + 1;
```

```
        ...
α   100 EMPL     5      ← i est une variable locale d'adresse 5
    102 EMPG    20      ← n est une variable globale d'adresse 20
    104 INF
    105 SIFAUX   ?
    107 EMPL     5
    109 EMPC     1
    111 ADD
    112 DEPL     5
    114 SAUT   100
β   116
```

Production de code

- Exemple: compilation d'une boucle

```
WHILE i < n DO
    i = i + 1;
```

```
        ...
α   100 EMPL     5      ← i est une variable locale d'adresse 5
    102 EMPG    20      ← n est une variable globale d'adresse 20
    104 INF
    105 SIFAUX 116
    107 EMPL     5
    109 EMPC     1
    111 ADD
    112 DEPL     5
    114 SAUT   100
β   116   ...
```

Production de code

Boucle. Analyseur :

```
void instructionWhile() {    /* ici UC = WHILE */
    UC = uniteSuivante();
    expression();
    if (UC != DO)
        erreur("do attendu");
    UC = uniteSuivante();
    instruction();
}
```

instructionWhile →
WHILE *expression* **DO** *instruction*

Production de code

Boucle. Analyseur :

```
void instructionWhile() {    /* ici UC = WHILE */
    UC = uniteSuivante();
    expression();
    if (UC != DO)
        erreur("do attendu");
    UC = uniteSuivante();
    instruction();
}
```

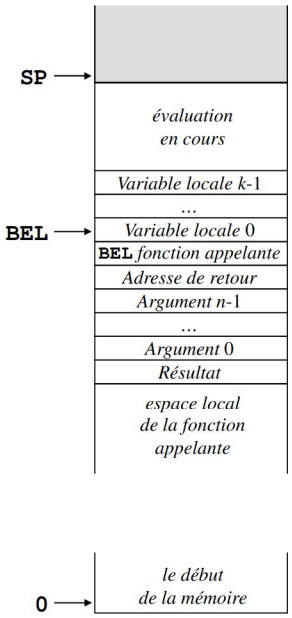
Compilateur :

```
void instructionWhile() {
    int debut, incomplet;
    debut = iMem;
    UC = uniteSuivante();
    expression();
    if (UC != DO)
        erreur("do attendu");
    UC = uniteSuivante();
    mem[iMem++] = SIFAUX;
    incomplet   = iMem++;
    instruction();
    mem[iMem++] = SAUT;
    mem[iMem++] = debut;
    mem[incomplet] = iMem;
}
```

```
    ...
α   expression
    ...
    SIFAUX β
    ...
    instruction
    ...
    SAUT α
β   ...
```

Production de code

Zoom sur le haut de la pile :
l'espace local d'une fonction



Comment depuis la fonction on accède…

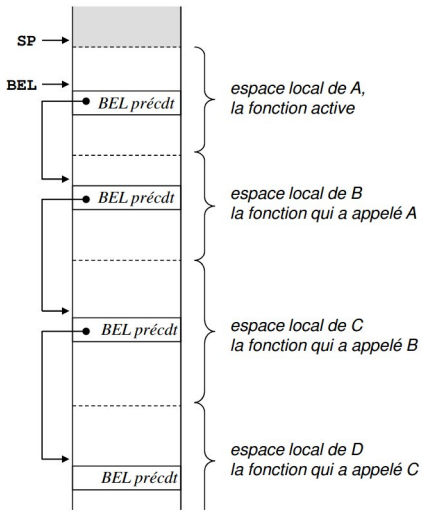
… à la $i^{ème}$ variable locale:
`BEL + i`

… au $i^{ème}$ argument:
`BEL - (n+2) + i`

… au résultat de la fonction:
`BEL - (n+3)`

Production de code

Chaînage des fonctions



Production de code

Code produit pour une fonction

```
int f(int a, int b, int c) {
    int u, v;
    ...
    return expr;
    ...
}
```

point d'entrée → `ENTREE` ; nombre de variables locales → `PILE 2`

```
...
code pour expr
DEPL -6
SORTIE
RETOUR
...
```

`DEPL -6` ← −(nbreargs+3)

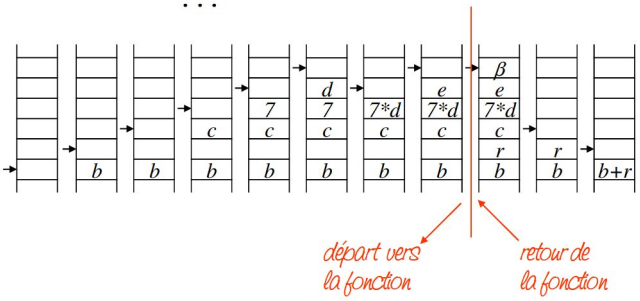
Production de code

Déroulement de l'appel d'une fonction. Exemple :

`… b + f(c, 7 * d, e) …`

(`b`, `c`… variables globales d'adresses 1, 2…)

```
        EMPG 1
        PILE 1      une place pour le résultat
        EMPG 2
        EMPC 7
        EMPG 3      on empile les arguments effectifs
        MUL
        EMPG 4
        APPEL α     l'appel
    β   PILE -3     on vire les arguments
        ADD
        ...
```

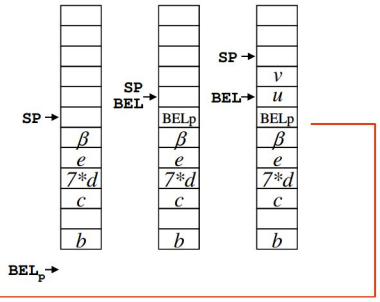


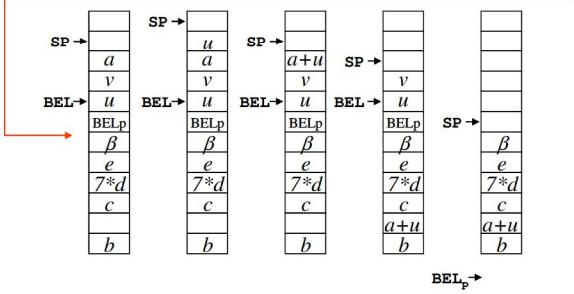
Production de code

L'intérieur de la fonction

```
int f(int a, int b, int c) {
    int u, v;
    ...
    return a + u;
}
```

```
α  ENTREE
   PILE 2
   ...
   EMPL -5
   EMPL 0
   ADD
   DEPL -6
   SORTIE
   RETOUR
```





Production de code

Le cas de la compilation séparée

- idents "publics" } *selon qu'ils sont accessibles*
- idents "privés" } *depuis les autres modules*

attention, ces mots n'ont pas le même sens que dans la POO

cela ne concerne que les "objets" :

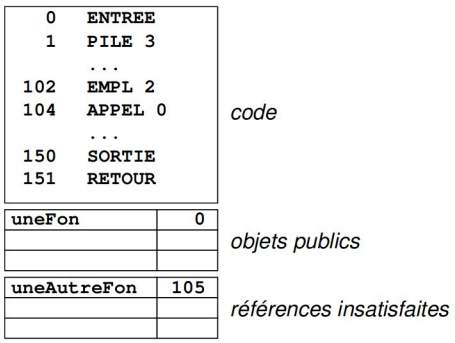
- constantes } *ne laissent rien dans le code*
- macros
- structures et types
- déclarations
- défs de variables globales } *on doit gérer le partage inter-fichier*
- défs de fonctions

Production de code

Fichier source

```
void uneFon() {
   int u, v, w;
   ...
   uneAutreFon(w);
   ...
}
```

Structure du fichier objet



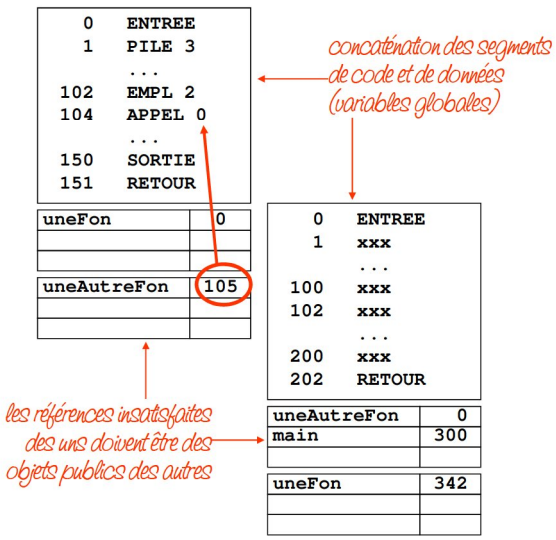
code

objets publics

références insatisfaites

Production de code

Edition de liens



concaténation des segments de code et de données (variables globales)

les références insatisfaites des uns doivent être des objets publics des autres

Le travail est le même

- pour les fonctions
- pour les variables (globales)