BABYAPL

Mini Interprète APL

sous Windows

Par Roger Busi

Déroulement de la présentation

I **Présentation personnelle**

II **Un interpréteur APL nommé BabyApl**

- Démonstration
- Mécanismes d'interprétation
- Bilan, Suite à donner

III **Questions sur le projet BabyApl**

Présentation personnelle

- **CV**
- **ACONIT**
- **Projets APL**

Contexte personnel / CV

Ingénieur ESME 90 (option Mécanique-Electricité)

Spécialiste en coupe des métaux (usinage)
Chargé du labo d'essais de coupe au **CTDEC** (Centre Technique du Décolletage - 74) – Pilote projet « Coupe » du Pôle de Compétitivité de La Vallée de l'Arve

Centres d'intérêts en informatique :
Fonctionnement des **interpréteurs et compilateurs** :
- Structurés : ALGOL, PL/1
- Non structurés : BASIC, FORTRAN
- Spéciaux : Forth, APL

Processeurs cablés ⇔ histoire de l'informatique
Interfaçage avec un processus (E/S) & **Simulateurs**

Contexte personnel / ACONIT

Membre de l'ACONIT (Grenoble) :
Association pour un **CON**servatoire du patrimoine de l'**I**nformatique et de la **T**élématique

Adhérent depuis 2003 => **Récupération matériel & doc.**

A savoir : sur le matériel « APL » récupéré par l'association :
- IBM 1130 avec clavier APL
- IBM 5100
- MCM (modèle ?)
- QL & ATARI & documentation de G. Langlet

Et un petit bonjour de **Philippe Denoyelle** aux « anciens » !

Contexte personnel / Projets APL

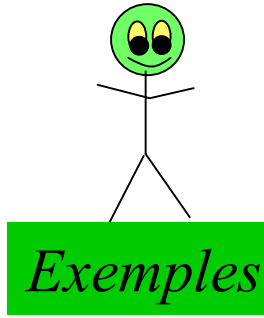
Suite aux cours que j'ai donné au CNAM sur les compilateurs (de 92 à 96) :

- **Rome 1998** : BABAPL calculatrice APL (Windows)
- **AFAPL 2000** : Mini interprète APL sur PSION
- De 2000 à 2004 : Réflexions sur une réalisation de type calculette APL (hard spécifique)
- De 2004 à 2006 : Portage sous Windows (16 bits) …
- **AFAPL 2006** : Mini interprète APL sous Windows (Basic 32 bits)

Interpréteur BabyApl

- **Démonstration**
- **Mécanismes d'interprétation**
- **Bilan, Suite à donner**

Démonstration



- **Tracé de graphiques**
- **Développement en série de Fourier**
- **Del editor**
- **E/S fichiers => Import/Export vers EXCEL**

Mécanismes d'interprétation

- **Hypothèses de travail**
- **Compilateurs vs Interpréteurs**
- **Analyseur lexical & syntaxique, et BNF**
- **Interpréteur de commandes**
- **Table des Symboles**
- **Analyse des expressions**
- **Pile de données symbolique**
- **Primitives APL**
- **Appel de fonction utilisateur, et retour**
- **Contrôle d'éxécution**
- **E/S (caractères, graphiques et fichiers)**
- **Gestion de la mémoire**

Hypothèses de travail / BabyApl

Les choix :

•Environnement **Windows** 98 ou supérieur

•Langage de programmation interprété/compilé en mode **32bits**

•**Mode Texte** avec sorties graphiques

•Pas de police de caractère APL, mais emploi de **caractères ASCII (cf APL11)**

•**Simplicité** :

⇒Outil de calcul simple pour l'ingénieur

⇒Niveau de fonctionnalités minimal

⇒Export données vers EXCEL

Hypothèses / Langage de développement

Culturellement, il y avait le choix entre :

- **BASIC** (Quick Basic,Visual Basic)
- **C** …

Choix **initial** : Visual Basic 1.0 (limitations 16 bits…)
Puis **dernièrement** : BBC Basic (peu de limitations)

Migration du source en VB vers BBC Basic en Février 2006

Hypothèses / Police de caractères

Idée initiale : APL11 sur Vax et QL

Utilisation des **caractères ASCII** :

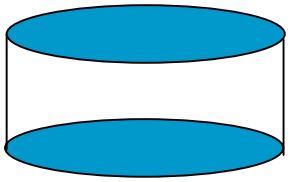
⇒Minuscules pour les noms de variables

⇒Majuscules pour les opérateurs

Conventions :

- Caractère sur la même touche que le caractère APL : **2 3RI6**
- Caractère(s) ressemblant(s) : **L** ⇔ quad
- Caractère(s) restant(s) disponible(s) : **{** ⇔ affectation
- Combinaison ressemblante : **O** ⇔ transpose

Langages / Compilateurs

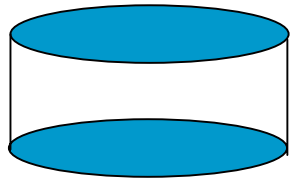


Source

Compilateur

1 ou plusieurs passes

- Analyseur lexical
- Analyseur syntaxique
- Analyseur sémantique
- Générateur de code intermédiaire
- Optimiseur de code intermédiaire
- Générateur de code



Objet

Mémoire programme

Mémoire données

Table des symboles*

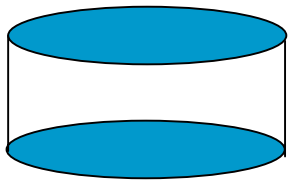
Pile d'exécution*

Pile des retours

**Table des symbole : PL/1*

**Pile d'exécution : machines à pile*

Langages / Interpréteurs



Source

Interpréteur

- Analyseur lexical
- Analyseur syntaxique
- Analyseur sémantique
- Fonctions primitives

Table des symboles

Mémoire programme

Mémoire données

Pile d'exécution

Pile des retours

- Interpréteur de commandes
- Editeur de texte

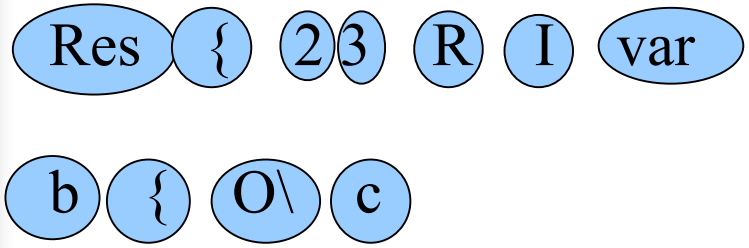
Analyseur lexical

Objectif :

Rechercher des **unités lexicales (tokens)** dans une ligne de texte, et les passer à **l'analyseur syntaxique** (pré-traitement)

Res { 2 3 R I var
b { O\ c

Devient :



2 fonctions de base :

Getchar() retourne le car. suivant

Lexeme() retourne le lexème suivt

- Le type (cste num …)
- La valeur numérique (12)
- La chaîne de caractères («titi»)
- Le n° de primitive (Iota = n°201)

Et supprime les espaces

Analyseur Syntaxique / Notion de grammaire

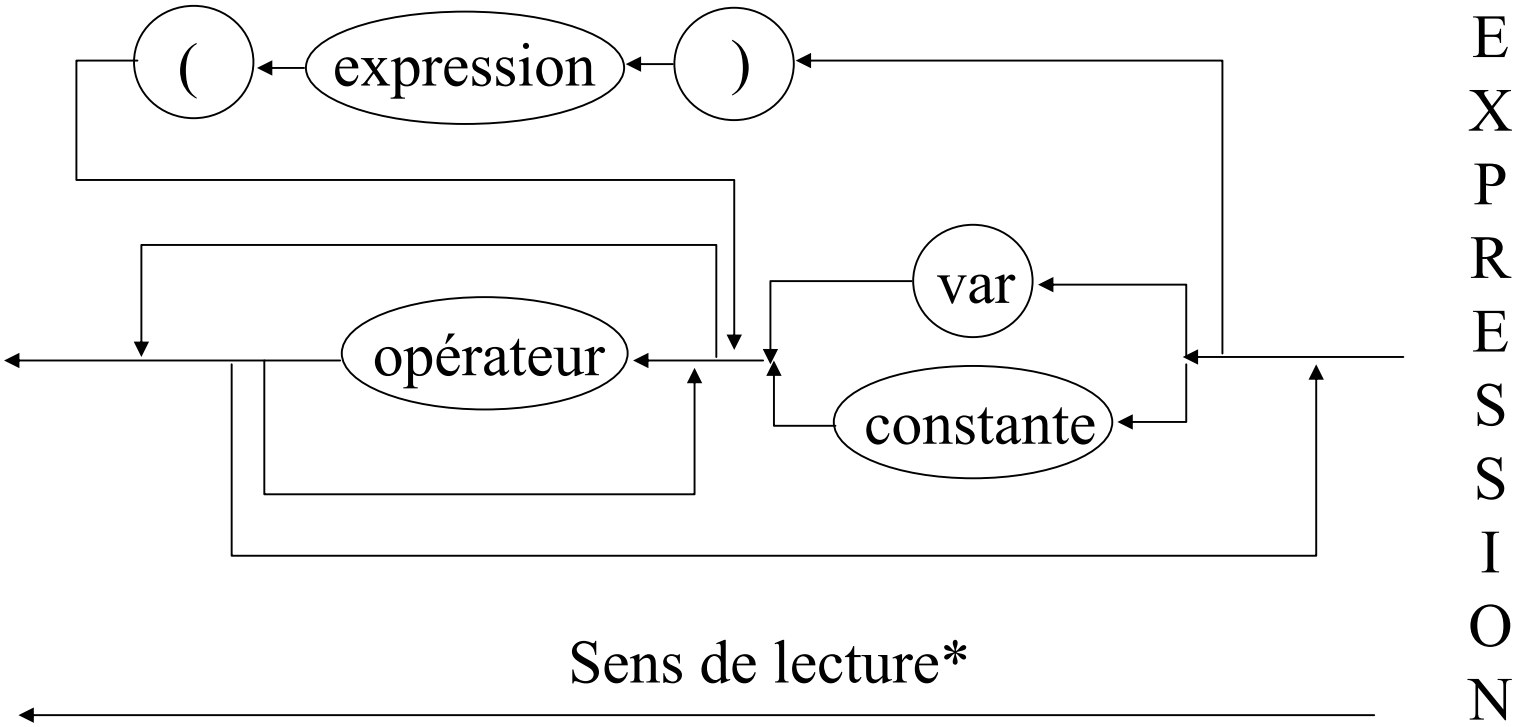
L'ensemble des différentes possibilités d'agencement des différents tokens forme la **grammaire du langage** qui doit être :

- Précise
- Non-ambiguë
- Exhaustive

Cette grammaire peut être décrite par :

- Des **diagrammes syntaxiques**
- La **notation BNF** (Backus-Naur Form)

Analyseur Syntaxique / Diagramme syntaxique



var{2 3RI(2+titi)

Analyseur syntaxique / BNF

Description d'une grammaire sous forme « texte » lue de gauche à droite (**sauf pour APL**)
Grammaire = suite de règles (règles de production)
C'est un assemblage valide de terminaux

Non-terminal ::= suite d'alternatives | comprenant
- Des terminaux (p.ex. +)
- Des non-terminaux (cf ci-après)
- Des parties facultatives […]
- Des parties optionnelles { …} (0,1 ou plusieurs fois)

Un non-terminal = variable intermédiaire de description

BNF / Exemple

Ex :

Nombre ::= chiffre { chiffre }

Chiffre ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Autre possibilité d'écriture :

Nombre ::= chiffre { chiffre }

Chiffre ::= 0

Chiffre ::= 1

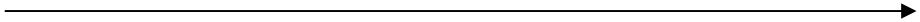
…

Chiffre ::= 9

BNF / Suite

« Une **grammaire dérive** des chaînes en commençant par l'**axiome** (symbole de départ), et en remplaçant de manière répétée un non-terminal par la partie droite d'une des production le définissant »

(sens de lecture*)



Nombre ::= chiffre { chiffre }
Chiffre ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Non terminal

Terminaux

BNF / Dérivation à gauche

Dérivation à gauche signifie :

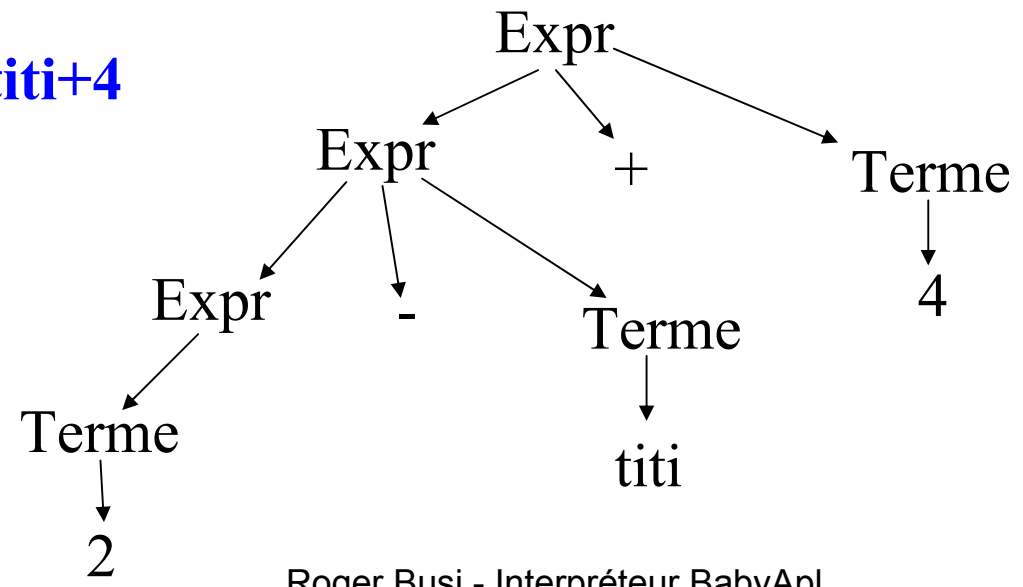
Expr ::= expr + terme

Expr ::= expr - terme

Expr ::= terme

Terme ::= nombre | identificateur

Ex : **2-titi+4**



BNF / Dérivation à gauche

Soit :

Expr ::= expr + terme

Expr ::= expr - terme

Expr ::= terme

Terme ::= nombre | ientificateur

⇒**Pb de la récursivité à gauche** (boucle infinie de l'analyseur)

Solution, il faut **transformer la BNF**, qui devient :

Expr ::= terme R

R ::= + terme

R ::= - terme

R ::= E (vide)

Terme ::= nombre | identificateur

BNF / BNF d'un mini BASIC

BNF d'un **mini BASIC** (extrait…) :

Chiffre ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Nombre2 ::= chiffre { chiffre }
Nombre ::= [+ | -] nombre2
Numéro ::= nombre2
Ident ::= A | B | C … | Z
Terme ::= ident | nombre
Expr ::= expr + terme
Expr ::= expr – terme
Expr ::= expr * terme
Expr ::= expr / terme
Expr ::= (expr)
Expr ::= NOT expr
…
Opcond ::= > | < | = | >= | <= | <>

BNF / BNF d'un mini BASIC

Suite :

Chaîne ::= « suitecar »

Texteprint ::= chaîne | expr

Affectation ::= ident = expr

Ligne ::= numéro REM suitecar

Ligne ::= numéro [LET] affectation

Ligne ::= numéro GOTO numéro

Ligne ::= numéro IF [expr] opcond expr THEN numéro

Ligne ::= numéro GOSUB numéro

Ligne ::= numéro RETURN

Ligne ::= numéro INPUT chaîne , ident { , ident }

Ligne ::= numéro PRINT texteprint { , texteprint } [;]

Ligne ::= numéro FOR ident = terme TO expr [STEP expr]

Lugne ::= numéro NEXT ident

Ligne ::= numéro END

BNF / BNF d'un mini BASIC

Programme ::= ligne L { ligne L }
Avec L ⇔ retour chariot
@@@@@ fin de la BNF (extrait) @@@@@

<u>Exemple de programme mini-BASIC :</u>

```
10 REM programme test
20 INPUT « Entrer un nombre » , A
30 IF A<=100 THEN 60
40 PRINT « nombre > 100 ! »
50 GOTO 90
60 FOR I=1 TO 4
70      PRINT « nombre <= à 100 ! »
80 NEXT I
90 PRINT « salut ! »
100 END
```

BNF / Choix de la production

Comment évaluer les règles de production ?

1) Par **essais-erreur** avec remontée si cela ne colle pas …

2) Par l'emploi d'un **symbole de pré-vision** (ex en BASIC) :

```
REM remarque
DIM tab(5)
FOR i=1 to 5
LET z=i*10
y=z/2
NEXT i
END
```

Décl-tab **::=** DIM ident (nb { , nb })

Il y a toujours un mot clé, sauf pour l'affectation.

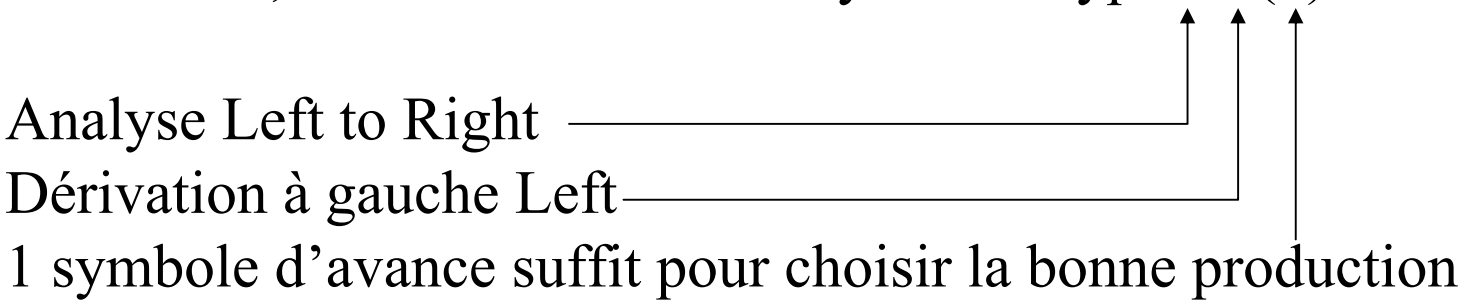
BNF / Emploi symbole de pré-vision

Avec un symbole de pré-vision, on peut éviter le rebroussement arrière (pb avec les compilateurs …).

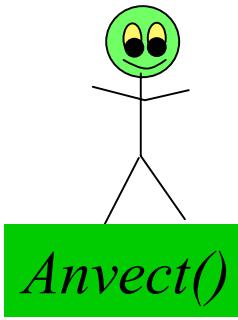
Il faut donc un **analyseur syntaxique prédictif**

⇒**Traduction dirigée par la syntaxe**

En BASIC, on a recours à un analyseur de type LL(1)



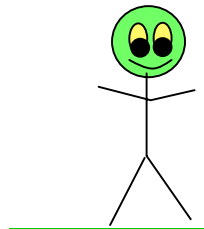
BNF / BNF BabyApl



BNF* BabyApl (lecture BNF de droite à gauche) :

Nb2	::= 0 | 1 … | 9 | E | _ | .
Nb	::= [_] nb2
Carmin0	::= a | … | z | 0 | … | 9
Carmin	::= a | … | z
Car	::= (tous les caractères du clavier)
Ident	::= carmin { carmin0 } carmin0
Vecteur	::= { nb } nb
vectalpha	::= ‘ { car } car
Label	::= ident :

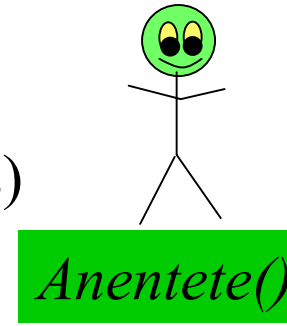
BNF / BNF BabyApl

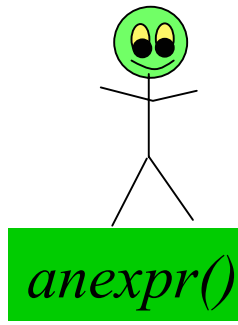


Suitevarindex ::= [;] E
Suitevarindex ::= suitevarindex ; expr
Varindexée ::= ident «[« suitevarindex «] »
Lignecli2 ::= [argg {] expr
Suitelignexe ::= }
Suitelignexe ::= argg {
Lignexe2 ::= [label] suitelignexe expr
Lignecli ::= { lignecli2 & } lignecli2
Lignexe ::= { lignexe2 & } lignexe2
Argg ::= L | Lg | Lg3 | ident | varindexée «] »
Argg ::= Lfile | Lf | Lts
Argd ::= vecteur nb | vectalpha ‘ | (expr) | ident | varindexée «] »
Argd ::= L | Ls | Lf

BNF / BNF BabyApl

Entetefn ::= [resultat {] ident { ; ident }
Entetefn ::= [resultat {] ident ident { ; ident }
Entetefn ::= [resultat {] ident ident ident { ; ident }
Entetefonction ::= << entetefn
Finfonction ::= >>
Op ::= I | O\ | … (cf liste des opérateurs)
Suitexpr ::= E
Suitexpr ::= op
Suitexpr ::= op « [« oprang «]" »
Suitexpr ::= identfn
Suitexpr2 ::= argd J.
Suitexpr2 ::= argd op JJ
Suitexpr2 ::= argd
Anexpr ::= { [suitexpr2] suitexpr } argd





BNF / Caractères & Primitives

$	1	EOL BOL	S	106	1 Ceiling 2 Maximum
&	11	Diamant	O*	107	1 Ln 2 $Log_a b$
+	100	1 (rien) 2 Plus	*	108	1 Exp 2 Power
-	101	1 CHS 2 Minus	O	109	1 Pi times 2 Trigo&Sqr
X	102	1 Sign 2 Times	^	110	2 And
L%	203	1 Domino 2 Domino	V	111	2 Or
%	103	1 Inverse 2 Divide	^~	113	2 Nand
!	104	1 Abs	V~	114	2 Nor
D	105	1 Floor 2 Minimum			

BNF / Caractères & Primitives

~	112	1 not	,	250	1 Ravel 2 Catenate
>=	116	2 Greater or Equal	R	204	1 Shape 2 Reshape
<=	117	2 Less or equal	OI	253	1 Reverse 2 Rotate
=	115	2 Equal	I	205	1 Iota 2 Indexof
<>	121	2 Not equal	/	255	1 Reduce 2 Compress
>	118	2 Greater than	O\	206	1 Transpose 2 Transpose
<	119	2 Less Than	\	254	1 Scan 2 Expand
?	120	2 Deal	H	251	1 gradeup

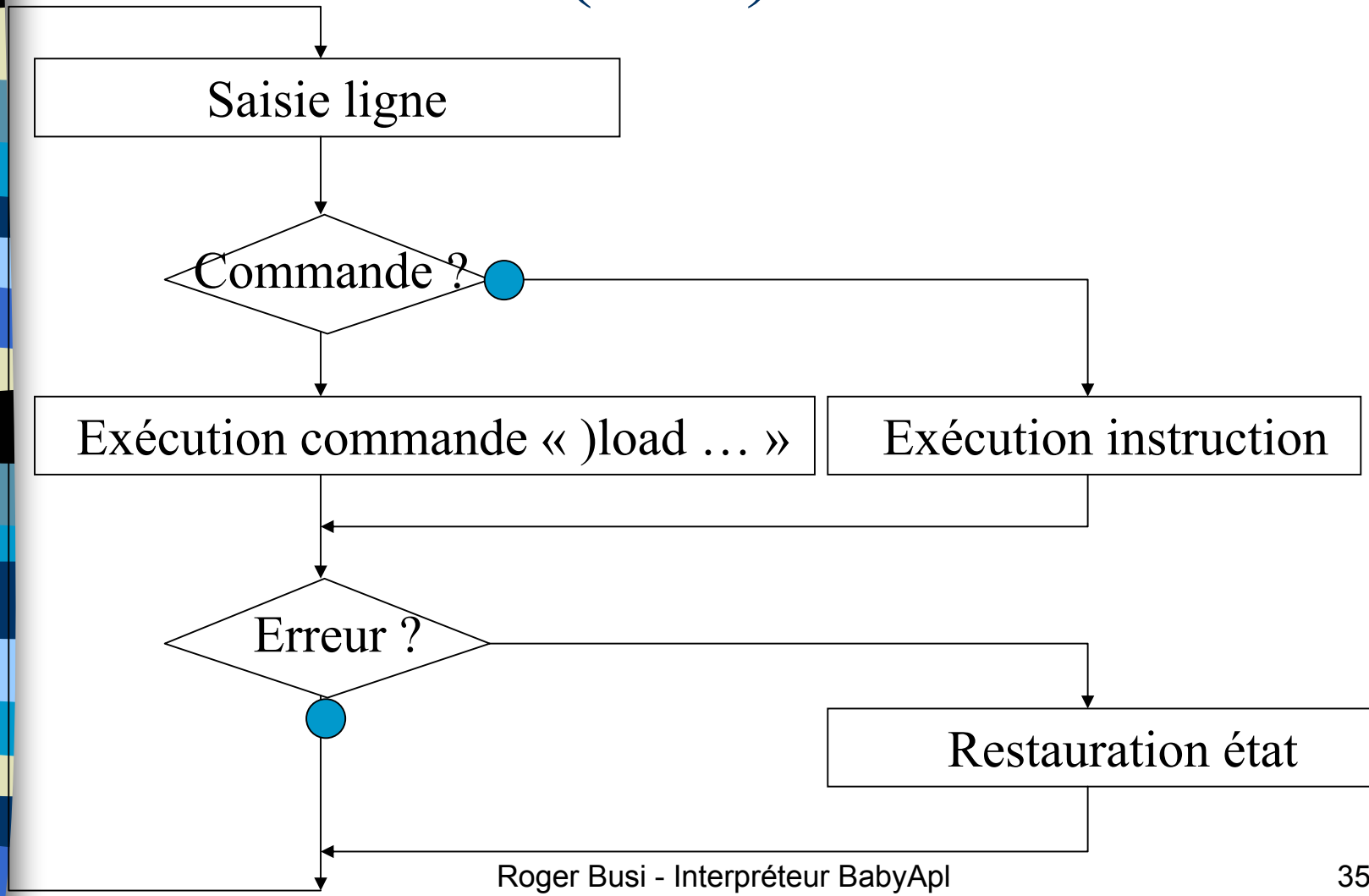
BNF / Caractères & Primitives

G	252	1 gradedown	[	2004	
Y	207	2 Take	]	2005	
U	208	2 Drop	(	2006	
M	256	2 Member	)	2007	
B	258	2 decode	;	2008	
T	259	2 code	J.	2009	1 Outer product
{	2000	Affectation	JJ	2010	2 Inner product
}	2001	jump	‘	2011	Chaîne de caractères

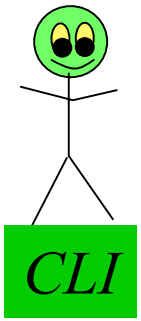
BNF / Caractères & Primitives

<<	2012	Début de fonction			
>>	2013	Fin de fonction			
]”	2003	Opérateur de rang			
L		Quad entrée Quad sortie	Lfile		Nom du fichier
Lg		Quad graphique 2D (courbes)	Lf		Quad file in Quad file out
Lg3		Quad graphique 3D(surfaces)	Lts		Affiche la Table des Symboles

BabyAPL / Interpréteur de commandes (CLI)

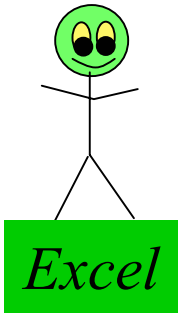


CLI / commandes



)off : sortie de BabyApl
)clear : efface le WS (TS + variables + programme)
)load nom : charge un WS (« programme »)
)save [nom] : sauve un WS (« programme »)
)erase nom : supprime une variable
)tron : mode trace actif
)troff : supprime le mode trace
)fns : liste des fonctions
)vars : liste des variables
)status : donne l'occupation (TS, mém. Progr., mém. données)
)systs : affiche la Table des Symboles
)sysdump adresse : fait un dump mémoire (20 nb depuis adresse)
)edit nom : appelle le « Del Editor »

Interprète BabyApl / Vue globale

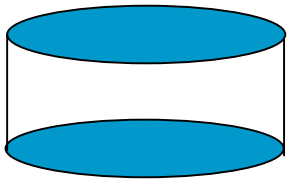


L'**interprète BabyApl** c'est :

- Un tableau qui simule la mémoire de données
- Un tableau qui simule la mémoire de programme
- Des fonctions de traitement (analyseur lexical, syntaxique, …)
- Un interprète de commande & une interface homme/machine

Cf schéma général des données sous EXCEL
Cf schéma général des fonctions sous EXCEL
Cf schéma général des appels de sous-programmes sous EXCEL

Interprète BabyApl / Vue globale



Source

- Analyseur lexical
- Analyseur syntaxique
- Analyseur sémantique*
- Primitives APL

Table des symboles

Mémoire programme

Mémoire données

- Gestion E/S
- Accès objets DATA
- Gestion TS
- Gestion pile DATA
- Gestion pile des retours
- Interpréteur de commandes
- Editeur de texte

Pile d'exécution

Pile des retours

BabyAPL / Table des symboles

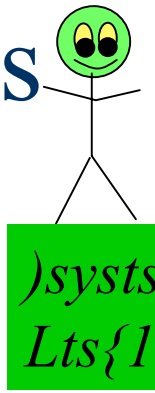


Table contenant différents champs :

nom$ /	type	/ adresse	/ autre-info
Var	/ VGlobale	/ 10000	/ 0
Fonc	/ fonction	/ 20	/ 30
Loop	/ label	/ 22	/ 0
Loc	/ VLocale	/ 12000	/ 3
…			

2 fonctions de base :

TSchercher (nom$) => retourne un indice ou -1

TSinsérer (nom$) => retourne un indice ou -1

BabyAPL / Table des symboles

Problème du stockage des noms à la « queue leu leu » dans la TS
=> boucle de recherche (long !)

Comment accélérer le temps de recherche dans la TS ?

Emploi de la **technique du Hash-coding** (p.ex. en recheche)
X = Somme des codes ASCII du nom$
Y = X MOD nb-éléments-TS
Recherche à la position Y dans TS
Si nom$ est à cette position => OK nom$ trouvé
Sinon cf en ligne suivante

Si rebouclage jusqu'à la position Y => nom$ inconnu de TS

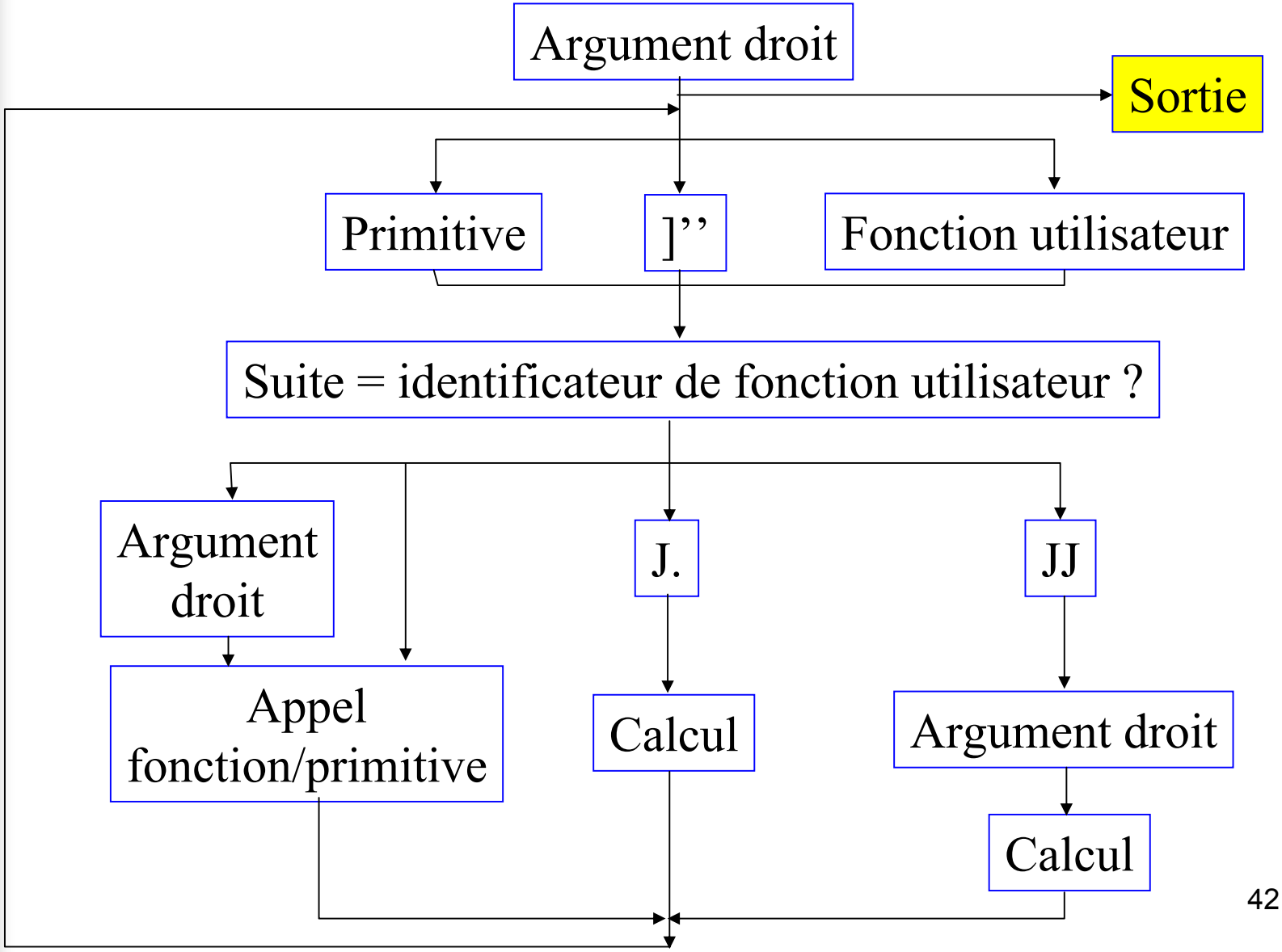
Interprète APL / mise à jour TS

Suite à un **)load** ou à la **sortie de l'éditeur**, il faut **mettre à jour** la Table des symboles pour disposer :

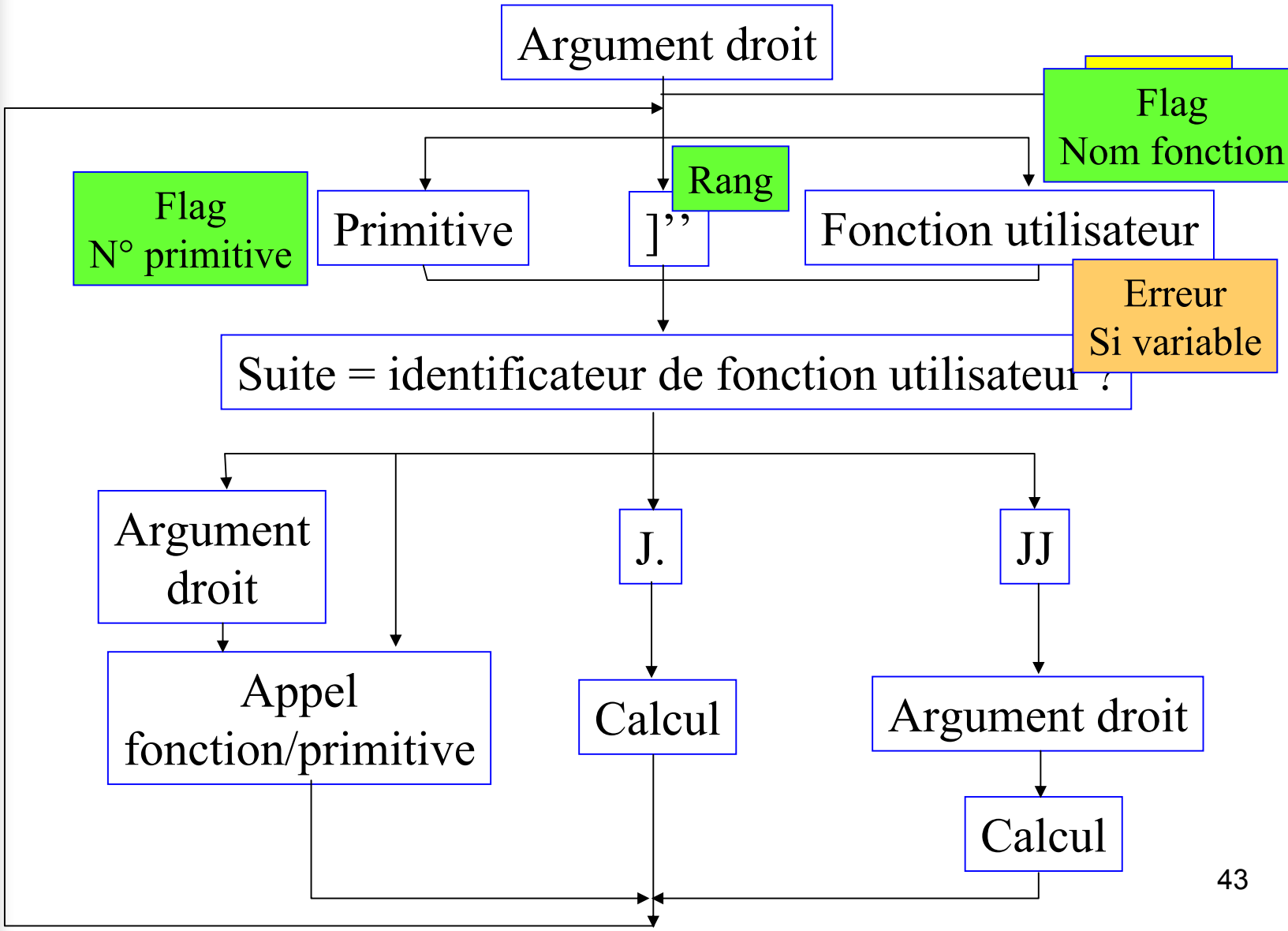
- Des **noms de fonction** (ainsi on pourra appeler rapidement une fonction sans lire tout le texte)

- Des **noms de labels** (tous différents*) (ainsi on pourra faire des sauts rapidement sans lire tout le texte)

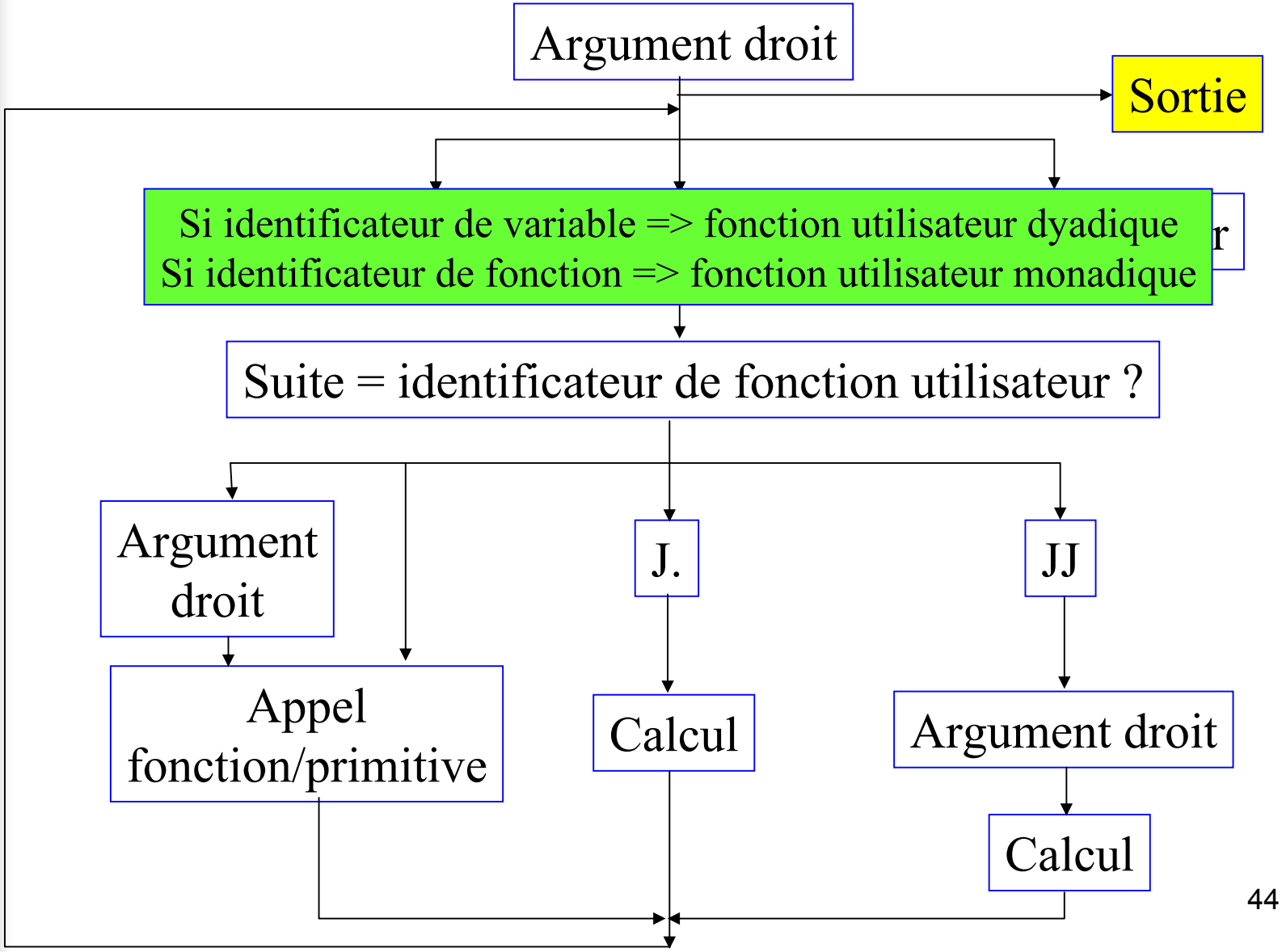
Interprète APL / Expressions



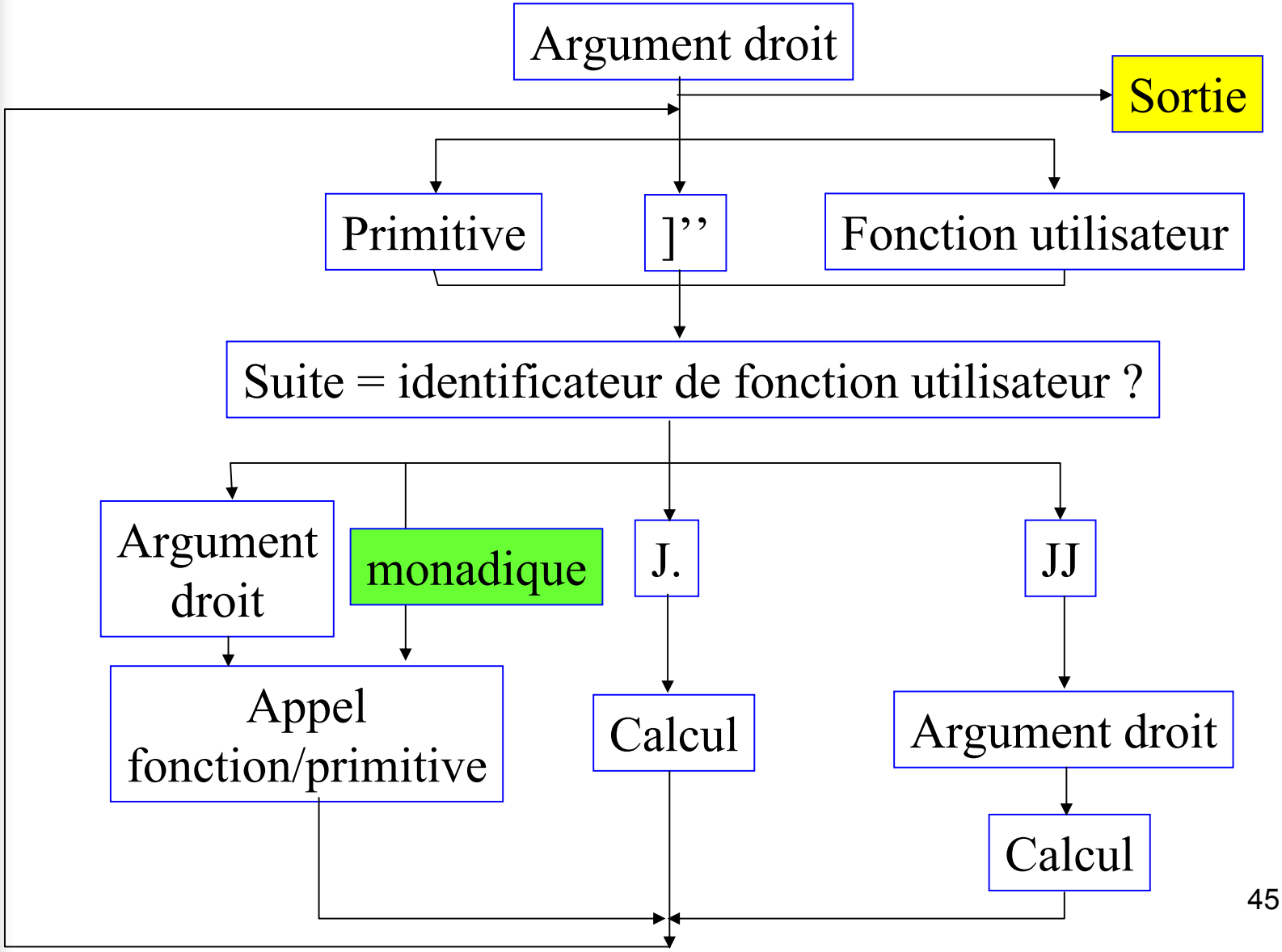
Interprète APL / Expressions



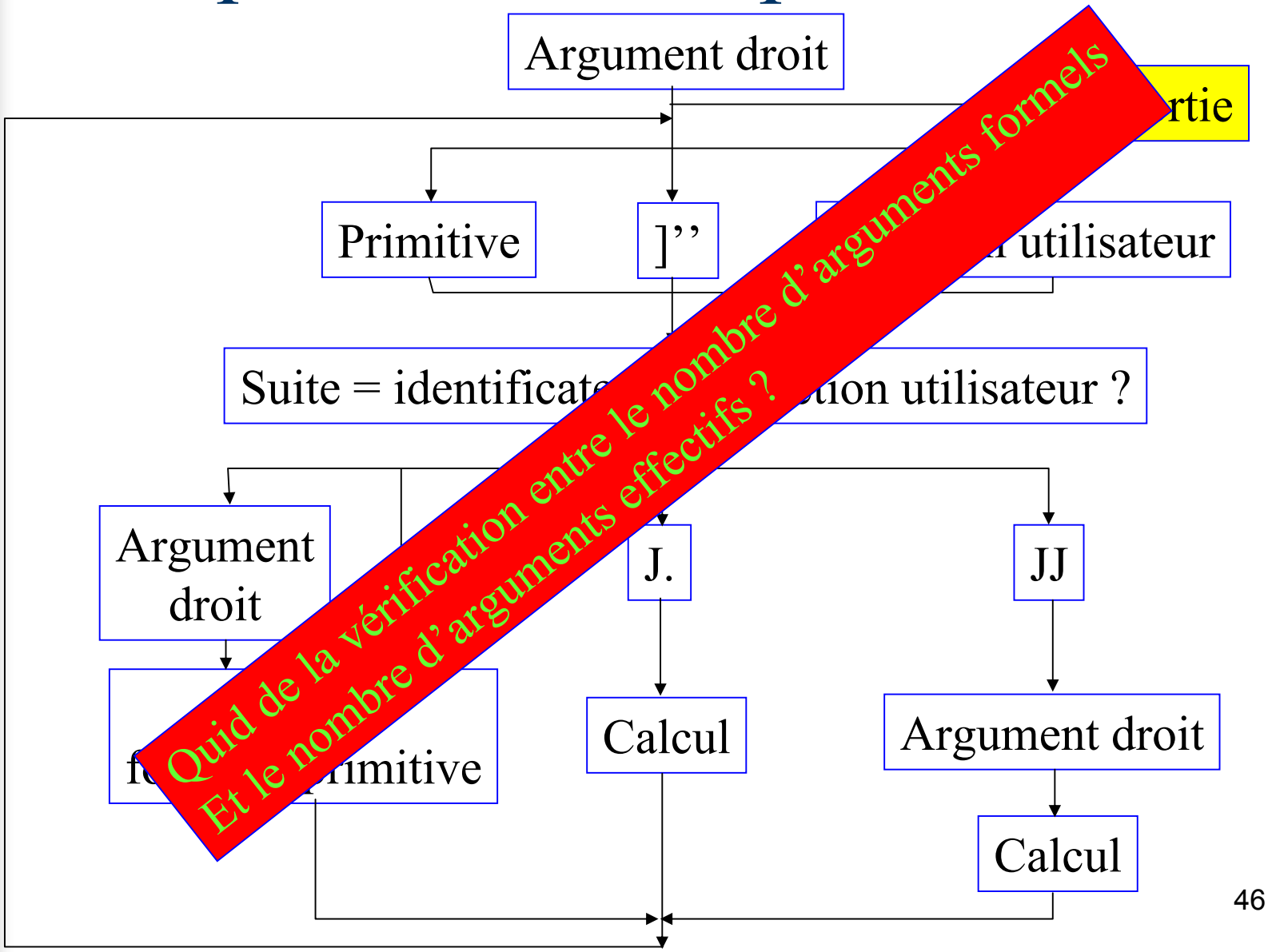
Interprète APL / Expressions



Interprète APL / Expressions



Interprète APL / Expressions



Interprète APL / Pile symbolique

Dans la suite de **BABAPL (Rome 98)** et dans la logique du **langage Forth** => utilisation d'une **Pile symbolique de données** :

- Toute variable est mise (complètement) sur la pile
- Toute opération (scalaire, mixte) se fait sur la pile

=> Les arguments de fonctions sont aussi sur la pile

La pile communique avec la mémoire de données :

Accès variable => mémoire DATA => mise sur pile
Affectation => stockage pile => mémoire DATA

Les données sont des **objets** qui contiennent :

- Un « **header** » : entête (rang, nbi, nbj, nbk)
- Un « **body** » : contenant nbi x nbj x nbk **réels**

Interprète APL / Pile symbolique

Il existe des objets scalaires, vecteurs ou tableaux (éventuellement vides).

Choix de programmation (très criticables !!!) :

- **Pas de types (réel, entier, booléen, chaînes de caractères)**
- **Toujours les 3 dimensions dans l'entête (nbi, nbj, nbk) acceptable pour 3 dimensions maxi !**
- **Pas d'objet pointeur (pb : place sur la pile par duplication)**

  Ainsi : **tableau{tableau+constante**

  Il y a **temporairement** 2 fois le tableau sur la pile,
  et 1 fois dans la zone mémoire après la copie !!!

Interprète APL / Manipulation de pile

Fonctions de **suppression du sommet et du ou des sous-sommets de pile** :

- Gcs : efface le sommet de pile
- Gcss : efface le sous-sommet de pile
- …

Fonctions de **manipulation de l'entête** :

- Newheader : crée un entête
- Delheader : supprime un entête
- Setheader(rang,nbi,nbj,nbk) : initialise l'entête

Fonctions de **lecture/écriture** dans les objets sur pile:

- Items(i,j,k) : lit un nombre
- Setitems(i,j,k,x) : écrit un nombre dans le sommet
- Setitemss(i,j,k,x) : écrit un nombre dans le sous-sommet

Interprète APL / Opérateurs scalaires et mixtes

Un **dispatching** appelle le sous-programme adéquat.

La structure globale d'une **fonction primitive APL** est la suivante :

- **Vérification** du bon nombre d'arguments
- **Création d'un nouvel objet** résultat (vide) sur la pile
- **Définition de ses dimensions** et initialisation de l'entête
- **Boucle générale** de parcours des arguments, et de calcul des éléments du résultat
- **Suppression** du ou des sous-sommets(s) de pile inutiles

Spiota()
Opscalaire2()

Interprète APL / Opérateurs scalaires et mixtes

Difficultés rencontrées dans le codage des primitives :

Conditions aux limites p.ex.cas de la concaténation généralisée :

- x,y avec x et y de même rang
- x,y avec x et y de rangs avec une différence de 1

⇒ Gestion lourde des différents cas avant de faire la boucle de calcul

Cas de l'extension scalaire

Cas du vecteur par rapport à un tableau à 1 dimension

Choix « simplificateurs » :

- **⎕IO** n'existe pas, ainsi que d'autres …
- **Gestion du vecteur vide** non standard => «**Empty vector error**»
- Pas de **tableaux en indices**

…

Interprète APL / Entête fonction

BNF :

Entetefn ::= [resultat {] ident { ; ident }

Entetefn ::= [resultat {] ident ident { ; ident }

Entetefn ::= [resultat {] ident ident ident { ; ident }

Entetefonction ::= << entetefn

L'appel d'une fonction utilisateur entraîne la lecture de D à G de l'entête avec un **stockage dans une table temporaire** avec des index :

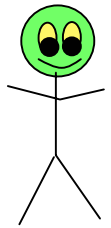
- Nombre de **variables locales**
- Indice sur **arg droit, nom fonction et arg gauche**
- Indice présence **arg résultat**

Interprète APL / Entête fonction

Puis la **création des nom$** dans TS :

- **Incrémenter le niveau** d'imbrication pour autoriser des appels récursifs p.ex. avec les mêmes noms de var locales.
- **Création Arg droit** + adresse/flag + niveau d'imbrication
- **Création Arg gauche** + adresse/flag + niveau d'imbrication
- **Création Arg résultat** + adresse/flag + niveau d'imbrication

Nom de fonction déjà connu et stocké par TSSP( )



Newcontext()

Interprète APL / Pile contexte

Exemple :

titi{…O\2 3fntoto var

…

<< res{a fntoto b;loc1;loc2

…

res{…

>>

Lors de **l'appel, il faut sauvegarder** :

- Le **compteur programme** (n°ligne)
- Le **pointeur sur le texte d'entrée**
- **Drapeau** présence argument formel gauche
- **Drapeau** présence argument formel droit
- **Drapeau** présence résultat formel de fonction

Interprète APL / Appel de fonction

Au début de l'exécution de la fonction :

- L'adresse de **l'argument droit pointe sur la pile** (p.ex. sous-sommet si il y a 2 arguments)
- L'adresse de **l'argument gauche pointe sur la pile** (p.ex. sommet si il y a 2 arguments)
- L'adresse de **l'argument résultat pointe sur rien** (sert de drapeau)

Lors de **l'affectation ultérieure à l'argument résultat,** le pointeur pointera sur la mémoire de données (le Heap) => le résultat est alors manipulable comme toute variable (locale).

Interprète APL / Retour de fonction

A la sortie de la fonction, il faut :

- Lire les drapeaux et **supprimer les arguments sur la pile**
- **Supprimer les variables locales**
- Mettre sur la **pile le résultat** s'il existe (par lecture du drapeau)
- Restaurer les **pointeurs sur l'analyseur lexical**
- Restaurer le **compteur programme, et le niveau d'imbrication**

Attention :

Actuellement, il n'y a **pas de vérification efficace** sur le nombre d'arguments formels de la fonction et le nombre effectif

=> Le seul moyen de s'en apercevoir est par manque d'objet sur la pile de données !!!

Interprète APL / Contrôle

Définition des labels :

...

Loop: var{…

…

} (expr)/**loop**

…

Les sauts sont locaux à la fonction (mais les noms de labels doivent être tous différents*) !

Possibilités :

} ⇔sortie de toutes les fonctions

}I0 ⇔on continue en séquence

} nombre ⇔ saut à la ligne nombre

} label ⇔ saut à la ligne du label

Interprète APL / E/S graphiques

Afin de simplifier l'utilisation, il existe des **sorties graphiques** :

- Lg{vecteur => série de points
- Lg{matrice de 2 lignes => nuage de points
- Lg{matrice de plus de 2 lignes => plusieurs nuages (couleur)

- Lg3{matrice => surface z=f(x,y)

A cette date, il resterait à finir :

Lg : tracer les échelles horizontales et verticales

Lg3 : colorier les carreaux « à la EXCEL »

Interprète APL / E/S fichiers

Pour utiliser des fichiers, il faut pouvoir :

- **Ouvrir un fichier** : Lfile{'nom'
- **Lire le fichier** : Var{Lf
- **Ecrire dans le fichier** : Lf{var

Choix : on lit/écrit en 1 fois dans une seule variable !

Le fichier d'entrée est en texte avec l'extension « .TXT »
Le fichier de sortie est en texte avec l'extension « .XLS »

Dans tous les cas, on a dans le fichier texte la structure suivante :

<header>
Nbi (tab) nbj (tab) nbk
<Data>
Nb (tab) Nb

Interprète APL / E/S caractères

Rappel :

Les tableaux n'ont qu'un seul type, ils ne contiennent que des réels.

Pour ne pas casser la structure du programme actuel => choix du stockage des chaînes de caractères sous forme de suites de nombres (codes ASCII).

Possibilités (limitées) :

Var{'salut !' <= **stockage chaîne** de caractères dans variable

Ls{var <= **impression chaîne** de car

L{var <= **impression codes ASCII**

Interprète APL / Affectation

Actions consécutives à une **instruction d'affectation** :

c{bX2 3RI6

Les actions sont :

1. **Objet constante** 6 sur la pile
2. **Calcul** => I6 sur la pile*
3. **Objet constante** (2 3) sur la pile
4. **Calcul** => (2 3)RI6 sur la pile*
5. **Variable b** chargée sur la pile
6. **Calcul** => bX(2 3)RI6 sur la pile*
7. **Allocation** zone mémoire dans le bas du Heap
8. **Mise à jour avec création** dans la Table des Symboles
9. **Copie** sommet de pile dans variable c (dans le Heap)
10. **Effacement** sommet de pile (Pile vide)

Interprète APL / Affectation

Cas du stockage dans une variable existante.

P.ex. **Tab{4 5Rtab** entraîne :

1. **Objet tab** (copie) sur la pile
2. **Objet constante** (4 5) sur pile
3. **Calcul** objet résultat = 4 5Rtab sur la pile
4. **Effacement** du sous-sommet et du sous-sous sommet*
5. **Stockage** dans mémoire (**Garbage Collecting systématique**)
6. **Effacement** sommet de pile

BabyApl / Garbage Collecting

Garbage Collecting => Position du problème

Données sur pile :

Les données sur pile sont automatiquement créées & effacées

- Soit par des appels à des fonctions gcs( ), gcss( ) …
- Soit par simple décrémentation du pointeur de pile

Les données en mémoire (dans le Heap) doivent être gérées ainsi :

Var{2+Var

BabyApl / Garbage Collecting

Plusieurs cas :

1) Si **Var n'existe pas** => insertion dans TS et copie de Var en bas du Heap => aucune modification majeure dans TS et pas de décalages dans le Heap

2) Si **Var existe déjà et résultat de même taille** => remplacement au même emplacement dans le heap => aucune modification dans TS et pas de décalages dans le Heap

BabyApl / Garbage Collecting

3) Si **Var existe déjà, et résultat de taille différente (en +/-) alors** avec l'hypothèse de ne pas émietter le Heap => il faut faire des décalages :

3.1 / **Suppression de l'objet Var** dans le Heap + **décalage** des autres objets qui sont en dessous de Var (adresses inférieures) et **mise à jour** des pointeurs dans la TS

3.2/ **Copie de l'objet Var** dans le bas du Heap avec mise à jour de la TS

Choix (très criticable !!!)

BabyApl / Garbage Collecting

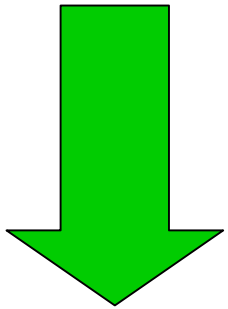
Autres options qui étaient possibles pour la gestion des « trous » dans le Heap :

- Choix de la **1ère zone** « trouée »
- Choix de la **zone « trouée » de taille la plus proche**
- Attente de **dépassement d'un seuil minimal** (distance) entre la pile symbolique et le bas du Heap avant de faire un Garbage Collecting

Ces options impliquent de conserver des pointeurs sur les objets inutilisés (table annexe de bits, ou dans la TS).

Bilan & Suite à donner

Description générale de l'interprète BabyApl donnée !



Bilan de ce projet
&
Suite à donner

Bilan

L'analyse de l'interprète actuel montre :

- Des **choix effectués criticables …**
- Un **besoin en simplification** au niveau des primitives APL
- Un **manque de fonctionnalités** (chaînes de caractères, E/S …)
- **Une relative lenteur** de l'interpréteur Basic actuel (comparé aux 1ères versions en VB)

Suite toujours dans l'optique **« calculette » de l'ingénieur** :

- **Stabiliser** la version actuelle => blindage au niveau de l'appel des fonctions …
- Améliorer le traitement des chaînes de caractères ? (**types**)
- Lire les ouvrages sur les interprètes APL (de M. Dumontier) pour **coder de façon optimale** les primitives APL, et réduire la taille du code source.
- **Tableaux de rang quelconques**.

Suite à donner

Question de fond :

Pertinence de continuer sur un développement personnel d'un interprète APL (cf offre Dyalog APL à 80Euros) … ?

Pertinence du développement d'une **calculette de poche APL** (non plus logicielle mais physique)?

- Hard spécifique (avantages/inconvénients)
- Hard existant (fabricants de machines à calculer)
- Reprise d'organiseurs de type Psion
- Emploi de pockets de type Palm

Fin de la présentation !