Travail d'Etude et de Recherche

Compilation et Expressivité d'un langage de programmation

Frédéric Gava

Maîtrise Informatique
Département Informatique
UFR des Sciences et Techniques
Université de Rouen
2000-2001

Je remercie
- Madame Léonard pour ses conseils sur les compilateurs et la grammaire du langage et ses encouragements
- M. Charras pour les technniques C de sauvegarde d'environnement
- M. Patrou pour les TP de Lex et Yacc.
- M. Lerest pour l'algo des huit reines.
- M. Duchamp pour ses encouragements.
- M. Valarcher pour ses conseils, idées, encouragements et le sujet de TER qu'il a proposé
- Et bien sûr, tous les professeurs d'Informatique de l'Université de Rouen pour la formation que j'y ai reçue.

Je remercie spécialement Hélène, pour les corrections Aurtografiks.

Sujet

Le sujet du TER était :

(1) Création d'un compilateur

(2) Implémenter un certain nombres d'algorithmes significatifs.

(3) Trouver les limites de ce langage.

(4) Trouver des améliorations significatives à ces limites.

Le sujet est donc mis théorique, mis pratique.

Pourquoi ce sujet ?

J'ai choisi ce sujet car je suis intéressé par la théorie de la programmation et de la calculabilité.De plus, le fait de pouvoir créer mon propre compilateur me permettrait d'avoir une gestion et un contrôle de l'ensemble du TER.

Introduction

La présentation des cours d'introduction à la programmation commence par la description d'un langage pseudo-algorithme, en général impératif. Ce langage est sans exception Turing-complet, c'est à dire avec une itération non bornée. A partir de cette boucle est construite l'itération bornée. Il est bien connu de la théorie des fonctions calculables que l'itération bornée ne permet de calculer que des fonctions qui se "terminent". Un langage de programmation n'ayant que des boucles bornées, permettrait la construction de programmes qui, toujours, terminent. Ce langage peut paraître trop peu puissant. Nous allons nous efforcer de montrer qu'il n'en est rien et qu'il permet de capturer un grand nombre d'algorithmes de base sans réelle perte de complexité.

La première partie est la description du compilateur et de la manière dont il a été construit. Nous décrirons donc l'analyse syntaxique et lexicale, puis la machine virtuelle et enfin la génération de code des instructions et des briques de ce langage.

La seconde partie est donc l'étude de ce petit langage du point de vue algorithmique plutôt que du point de vue extensionnel.

La troisième partie peut être vue comme une extension possible du langage. En effet, le langage décrit dans les deux premières parties ne permet pas de capturer toutes les algorithmes. Nous allons essayé de vous montrer comment remédier à ce problème, tout en gardant une terminaison sntaxiquement prouvée.

Table des matières

Table des figures

Première partie

Un petit compilateur

Introduction

Un compilateur est l'outil minimal pour pouvoir utiliser un langage de programmation. Celui-ci a été écrit en C ANSI, pour sa rapidité et sa très grande portabilité (tout système a au moins un compilateur C). L'analyse lexicale et syntaxique a été écrite en LEX et YACC pour leur simplicité d'utilisation et pour donner un code en C ANSI.

Ce compilateur ne peut être comparé à un "vrai" compilateur professionnel mais il a pour but de promouvoir notre petit langage et de permettre une utilisation facile et rapide pour tester les algorithmes.

Le premier travail d'un compilateur est l'analyse lexicale et syntaxique du fichier donné. On commence par une analyse lexicale pour déterminer et distinguer les mots clés des identifiants, ignorer les espaces et détecter les caractères inconnus. L'analyse syntaxique se chargera d'utiliser les lexèmes pour vérifier la cohérence syntaxique du fichier (vis-à-vis de la grammaire). Les tâches à accomplir (génération de codes intermédiaires, mises à jour des tables des symboles, erreurs) seront partagées dans les actions possibles des deux analyses.

Le code intermédiaire n'est pas très lisible humainement parlant. Par contre il est facile de l'exécuter sur une machine virtuelle (comme en Java). Dans ce travail le code intermédiaire n'est pas transformé en code machine (trop technique et pas portable) comme cela est fait dans la plupart des compilateurs.

CHAPITRE 1

L'analyse lexicale, l'analyse syntaxique et l'automate d'état

L'analyse lexicale et l'analyse syntaxique sont très liées l'une à l'autre (l'analyse syntaxique appelant l'analyse lexicale qui peut modifier l'état de l'analyse syntaxique). Décrivons maintenant les différentes parties de cette double analyse.

1. Les identifiants

Les identifiants sont les objets de notre langage. Ceux-ci sont divisés en trois catégories classiques de la programmation : les variables entières, les fonctions et les tableaux. Par simplicité, les règles de construction sont toutes distinctes, ce qui permet dès l'analyse lexicale de savoir à quelles catégories on a affaire (Certains langages comme CAML, PL-SQL utilisent cette technique pour d'autres catégories). L'action à effectuer pour ces catégories est le rangement de l'identifiant dans la table des symboles (ce rangement diffère bien entendu suivant l'avancement de l'analyse et le placement de l'identifiant dans le fichier).

1.1. Les variables. Les variables dans notre langage sont définies par les règles suivantes :
(lettres-majuscules)+(chiffre)* avec lettres-majuscules=[A..Z] et chiffres=[0..9]
Ainsi tout mot ne comportant que des lettres majuscules suivies d'un certain nombre de chiffres est une variable. Exemple : COUCOU,VAR1,VAR34,TMP1...

1.2. Les fonctions. La règle de construction des fonctions est le contraire de celle des variables. C'est une suite de lettres minuscules suivies de nombres :
(lettres-minuscules)+(chiffres)* avec
lettres-minuscules=[a..z]
Exemples :affiche,bonjour5...

1.3. Les tableaux. Les tableaux ont une règle un peu plus compliquée :
(lettre-majuscule)(lettres-minuscules)+(chiffres)*
Ce qui représente, une lettre majuscule, suivie d'un certain nombre de lettres minuscules et de chiffres. Exemple :Tab,Tub1,Toto45...

On peut constater que l'analyseur lexical ne peut pas se tromper entre les catégories. On remarque aussi que, bien entendu, la casse est à respecter.
Les nombres (>=0) sont aussi présents dans le langage. La règle est bien sûr :(chiffres)+. L'action peut différer s'ils sont utilisés pour la construction de tableaux ou dans les instructions d'affectation.

2. Les mots clés

Les mots clés de notre langage sont peu nombreux et écrits en minuscules. Ils permettent d'articuler le langage et donc de changer de partie lors de l'analyse du fichier.

2.1. Les mots clés du langage. Voici la liste des mots clés (la sémantique sera explicitée plus tard) :

- "debut", indique le début d'une fonction
- "fin", indique la fin d'un bloc (d'une boucle, hyperboucle, fonction)
- "creation", pour désigner qu'on va créer localement ou globalement des objets
- "entier", pour l'instant, sert à désigner qu'on va définir une fonction
- "boucle", indique qu'on va entrer dans une boucle
- "fois", maintenant, on rentre dans le code de cette boucle
- "hyperboucle", indique qu'on va entrer dans une hyperboucle

– "final", indique qu'on définit là le code "final" d'une hyperboucle
– "programme", pour indiquer qu'on définit maintenant le programme principal

Dans l'analyseur lexical, il faut détecter ces mots avant les fonctions sinon il y aura confusion. (debut,fin,fois,final) n'ont pas d'action spécifique mais ils servent à la bonne articulation du langage et à une meilleur lisibilité des sources pour un humain.

2.2. Les caractères spéciaux. Ceux-ci servent aussi dans l'articulation sémantique du fichier source, pour la lisibilité de ce même fichier ou pour des commentaires. Les actions sont toutes très distinctes et dépendent de l'avancement de l'analyse. Voici une liste de ces caractères :"tabulation","saut de ligne", "#", "+", "-", "=", " :", "(", ")", "[", "]", ";", ",".
Nous pouvons déjà décrire les actions de certains de ces caractères.

– "#" => indique un commentaire. La ligne du fichier source est ignorée de l'analyse. Il n'y a donc aucune action.
– tabulation," " => ce sont les séparateurs du langage. Aucune action à effectuer ici.
– " ;" => séparateur d'instruction. Aucun action à effectuer.
– " :" => Caractère "décoratif". Il n'a aucune influence sur le langage. Il permet une meilleure lecture des sources pour "creation :..." (voir la grammaire).
– "," => séparateur d'identifiants lors de la création de variables ou de tableaux ou lors des passages en paramètres de ces mêmes identifiants.

Les longs commentaires

Il est possible de commenter ses sources avec des commentaires sur plusieurs lignes. Les commentaires commencent par "/*" et ce terminent par "*/" (comme en C). On peut y mettre TOUT ce que l'on veut.
Comme ces commentaires sont sur plusieurs lignes, on pourrait utiliser le format LEX suivant : "/*"[.n]*"*/". Mais en procédant ainsi, on oublie de compter les lignes (les caractères n), ce qui n'est pas pratique pour l'affichage des erreurs. Il faut donc utiliser un boleens qui indique si on est dans un commentaire ou non (mis à vrai dès qu'on rencontre un "/*" et remis à faux dès qu'on rencontre un "*/"). Si les mots sont dans un commentaire alors l'analyseur lexical ne fait rien. Sinon, il fait son travail habituel.

Tout autre caractère est à bannir (sauf dans les commentaires). Si on en détecte un, il faut afficher un message d'erreur et arrêter la compilation.

3. Vision générale du langage

Il serait beaucoup trop long de décrire toute la grammaire du langage (voir Annexe). Nous allons plutôt en décrire les grandes lignes et montrer quelques exemples.
Le fichier source peut se décomposer en 3 parties :

(1) la création des variables globales et des tableaux globaux
(2) la définition des fonctions, revenir en 1
(3) le programme principal

3.1. Description des objets globaux. La première partie se fait tout simplement de la sorte :
"creation" " :" liste de créations variables et de tableaux
Cette liste comporte des noms de variables et des noms de tableaux ainsi que leurs tailles mises entre crochets. Exemples :
creation :TMP,Tab[23],Tub[5],VAR12

3.2. Définition des fonctions. La deuxième partie permet une construction incrémentale d'un algorithme. La définition des fonctions ressemble à celle du C. Voici la forme générale :
"entier" nom-de-la-fonction "(" paramètre ")"
"creation" " :" liste de créations variables et de tableaux
"debut"
suite-instructions
"fin"

"Les paramètres" est une liste de noms de variables et de tableaux. Les instructions seront explicitées plus précisement plus loin. Exemples :

```
entier coucou(A12,B,Tub)
creation:C,D,Tab[10]
debut
  ...
fin
```

3.3. Le programme principal. La troisième partie est la définition du programme principal. Elle ressemble beaucoup à celle des fonctions :

programme "(" paramètre ")"
"creation" " :" liste de créations variables et de tableaux
"debut"
suite-instructions
"fin"

Exemple :

```
programme(A,B)
creation:C,D,Tab[10]
debut
  ...
fin
```

Remarque : on n'est pas obligé de faire une "création" dans les fonctions et le programme principal.

3.4. Les modules. Les sources peuvent être écrites dans différents fichiers. Les fichiers ne contenant, que des définitions de fonctions, peuvent être vus comme des modules. Le dernier des fichiers doit bien entendu contenir le programme principal. Chaque module doit contenir une création d'objets globaux (même vide si on veut, voir grammaire). Le dernier fichier peut contenir soit le programme principal seul, soit des fonctions et le programme principal ou une création d'objets globaux suivie de fonctions et du programme principal.
Il est interdit de faire des créations d'objets globaux en milieu de fichier.

L'analyse de plusieurs fichiers, se fait en LEX grâce à la fonction "yywrap". Dans celle-ci, on ré-initialise la variable yyin sur le nouveau fichier à lire, on remet l'état en 6 et on met la variable de ligne courante en 1.

3.5. Appel de fonctions. Les problèmes indétectables par la syntaxe seront traités plus loin. Pour la suite nous allons tout de suite décrire l'appel des fonctions. Ceux-ci se font sur cette syntaxe :
VARIABLE " :=" FONCTION "(" parametres ")" ";"

Il est facile de remarquer que les identifiants n'ont pas le même rôle suivant qu'ils sont à créer, à définir ou qu'ils sont en paramètres d'une fonction. L'état de l'analyse change donc suivant qu'on est dans la définition des paramètres d'une fonction, dans le corps d'une fonction etc... et donc les actions à effectuer changent. Nous devons créer une variable d'états.

4. L'automate d'état

Les changements d'état peuvent être représentés par un automate.
- état 0 : création des identifiants de paramètres lors des définitions
- état 1 : création des identifiants d'une fonction
- état 2 : corps d'une fonction
- état 3 : identifiant mis en paramètres d'une fonction (VAR :=FUN(...))
- état 4 : identifiant d'une boucle (boucle VAR fois)
- état 5 : identifiant d'une hyperboucle (hyperboucle A fois)
- état 6 : création des identifiants globaux

Maintenant, on peut décrire les actions des identifiants suivant l'état

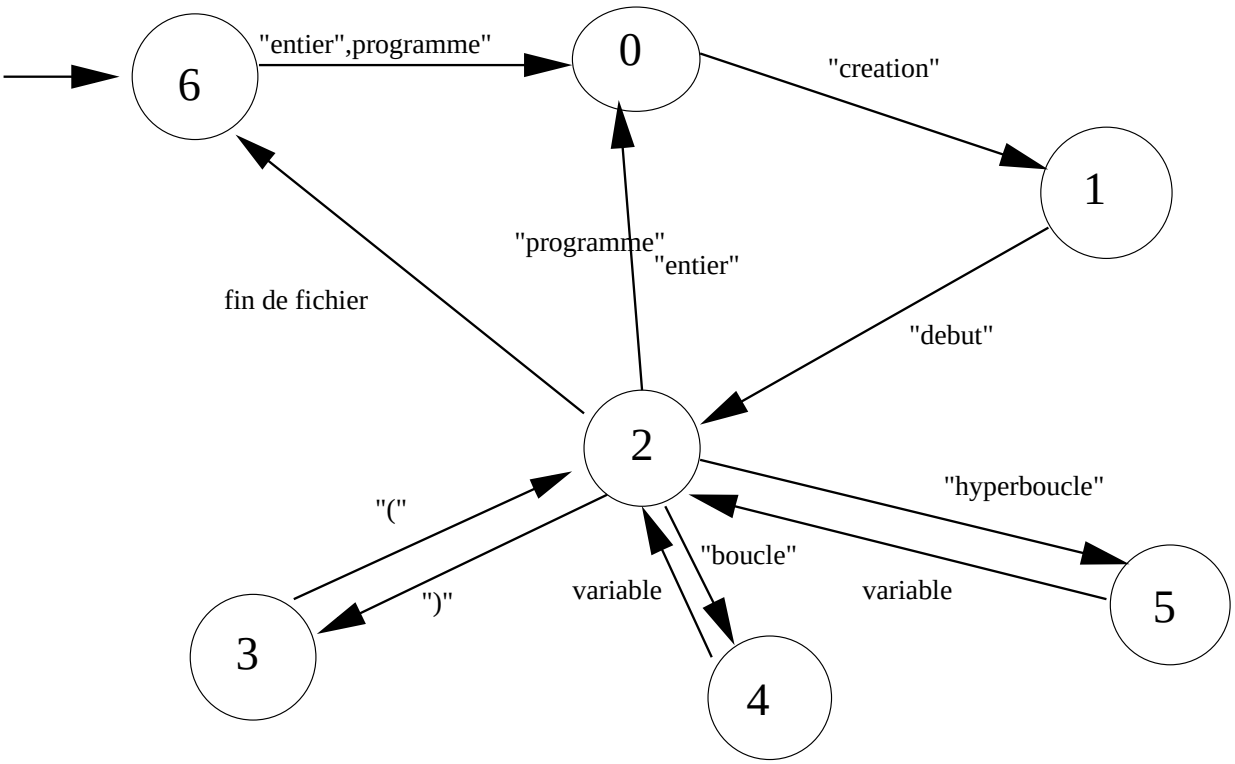

FIG. 1. L'automate des changements d'états

4.1. Action d'une variable. Algo

```
Si état=2 alors rechercher adresse de la variable dans la table des symboles
Si état=0 alors construction d'une variable en paramètre d'une fonction
Si état=1 alors création d'une variable locale
Si état=6 alors création d'une variable globale
Si état=3 alors ajout de la variable dans les paramètres en cours
Si état=4 alors code boucle état=2
Si état=5 alors code hyperboucle état=2
```

4.2. Action pour les tableaux. Le problème, pour les tableaux, est que lors de leur création, il faut attendre le crochet fermant pour connaître la taille. On utilise donc une variable de sauvegarde du nom du tableau et c'est lors du crochet fermant qu'on pourra effectuer l'action.

Tableaux

Algo

```
Si état=2 alors rechercher adresse du tableau
Si état=0 alors construction d'un tableau en paramètre d'une fonction
Si état=6 ou état=1 alors sauvegarder le tableau
Si état=3 alors ajout du tableau dans les paramètres en cours
```

Action après "]"

Algo

```
Si état=6 alors création d'un tableau global
Si état=1 alors création d'un tableau local
```

4.3. Action d'une fonction. Algo

```
Si état=0 on change de fonction dans la table des symboles
```

```
Si état=2 ou 4 ou 5 alors on mémorise le nom de la fonction pour la
génération de code.
```

5. Gestion des erreurs

Il y a trois types d'erreur :

(1) erreur lexicale : quand le lexeur trouve un caractère on arrête la compilation et on affiche un message d'erreur et la ligne.

(2) YACC est capable de détecter les erreurs syntaxiques, on affiche alors un message d'erreur, la ligne courante et le fichier.

(3) erreur sémantique : la liste est trop longue à énumérer mais pour toutes, on arrête la compilation, on affiche un message d'erreur spécifique à celle-ci et la ligne où se situe l'erreur.

Dans tous les cas, on arrête de compiler. Il n'est pas utile d'essayer de trouver le ou les caractères qu'il aurait fallu car on risque d'entraîner des affichages d'erreur en cascade (voir les compilateurs C et Java qui paniquent si on oublie un seul " ;").

CHAPITRE 2

Une machine virtuelle

Je vais maintenant décrire la machine virtuelle. Je le fais avant la génération de code pour bien comprendre le résultat obtenu.

Notre machine virtuelle, utilise un langage d'assemblage proche de celui des Intel pour PC. Notre machine virtuelle comporte une mémoire de 8000 entiers (dont 4 servant pour le système), une liste d'instructions, une pile et un pointeur d'instruction courante.

Le compilateur génère deux fichiers. L'un est exécutable (par la machine virtuelle), le second est une représentation de ce fichier en langage d'assemblage pour être lu par un utilisateur. En aucun cas ce fichier ne peut fonctionner sur la machine virtuelle. Il permet simplement un contrôle utilisateur. Les instructions dont nous allons parler seront celles du fichier exécutable donc avec leurs représentations en langage d'assemblage.

1. Fonctionnement général

En un premier temps, la machine virtuelle charge le programme en mémoire et place son pointeur d'instructions sur la première instruction. Pour cela on lit le fichier instruction par instruction et on teste la bonne écriture de ces instructions (instructions bien générées par le compilateur Lab) et on reitère l'opération jusqu'à la fin du fichier. La machine virtuelle lit instruction par instruction et ne s'arrête qu'en cas d'erreur ou d'une instruction de fin d'exécution.

La machine virtuelle ne fonctionne que sur des entiers ≥ 0.

1.1. Différents modes d'adressage. Il existe 3 modes d'adressage. Le premier, et le plus courant, est la lecture directe dans la mémoire (On demande la valeur de la ième case mémoire). On ne met pas de symbole devant le nombre (représentant une case mémoire). Le deuxième mode est l'adressage immédiat, c'est à dire que la valeur n'est pas une adresse mais une quantité entière. On met le symbole # devant le nombre pour la distinguer de la précédente. Le troisième, est le mode indirect. La valeur à recopier n'est pas la valeur contenue à l'adresse, mais le contenu d'une adresse elle-même contenue à l'adresse demandée.(voir figure). On met le symbole # devant pour distinguer le mode des deux précédents. Le troisième mode peut être vu comme un pointeur. Il servira pour le fontionnement des tableaux.

1.2. Utilisation. Les instructions sont de la forme :

```
instr adr1 adr2
```

Les instructions sont représentées par un enregistrement comportant :
- le type de l'instruction
- le type et la valeur du premier argument
- le type et la valeur du second argument

En C

```
typedef enum {PUSH,POP,ADD,SUB,MOV,MOD,ORG,END,INT,JMP,JZ,JNZ,JE} typeinstr;
typedef struct Instruction
 {
  typeinstr instr;
  int adr1;
  int type1;
```

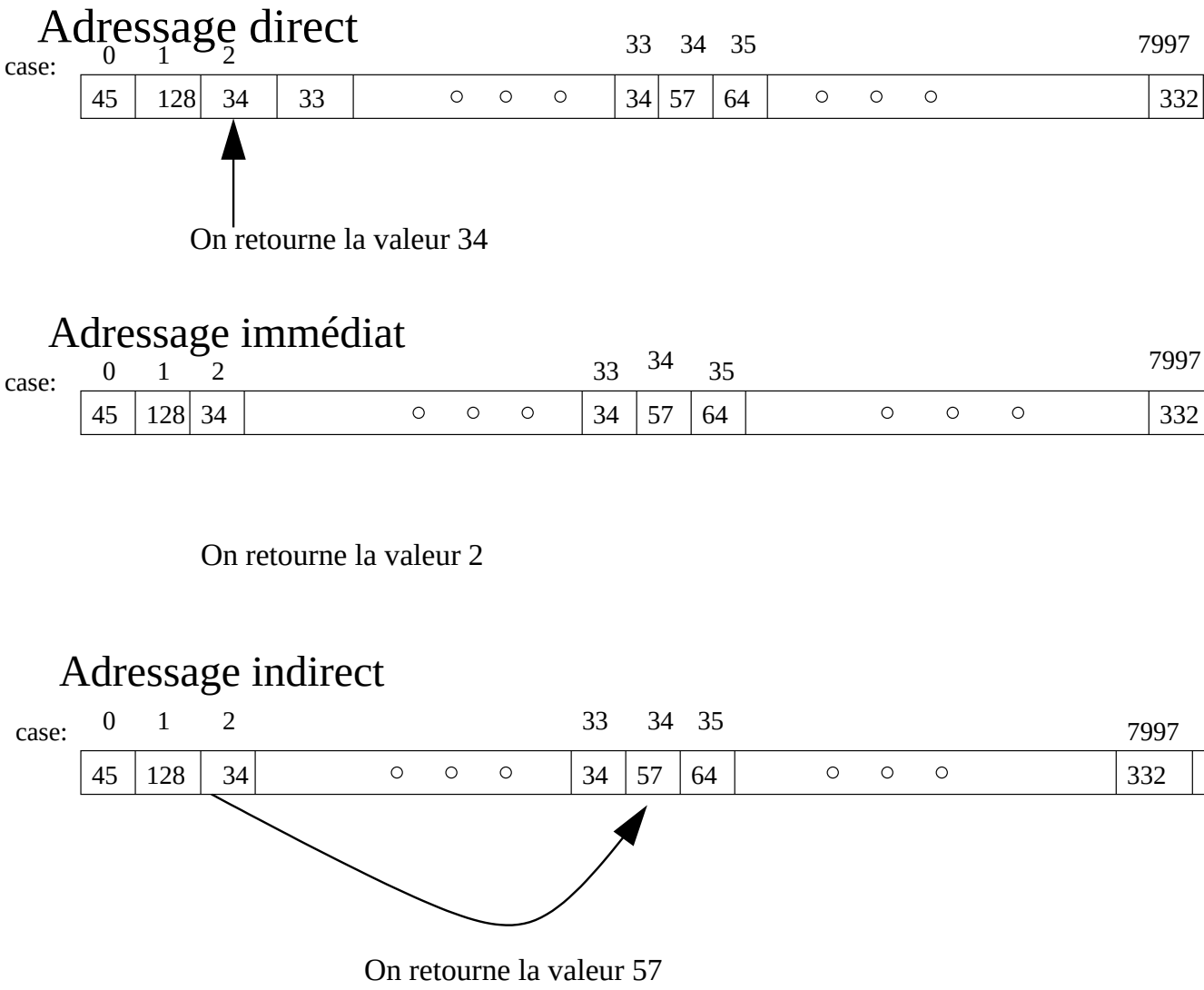

FIG. 1. Demande de valeur pour 2, #2 e 2

```
  int adr2;
  int type2;
} instruction;
```

La lecture est donc très simple à implémenter (lecture directe sur le fichier d'une instruction).

La lecture des valeurs suivant le mode est aussi très simple, Algo grossier
- cas 0, adressage direct, on retourne mémoire[adresse]
- cas 1, adresse immédiate on retourne adresse
- cas 3, adresse indirecte, si memoire[adresse] est trop grande, erreur sinon on retourne memoire[memoire[adresse]].

1.3. Algo général. Voici l'algo grossier du fonctionnement de la machine virtuelle. Nous supposons que nous pouvons récupérer les valeurs des paramètres des instructions. Nous verrons par la suite, les différentes instructions.

```
début
lire programme.
Empiler les arguments du programme.
fin_exe=faux;
```

```
pointeur=0;
tant que (not fin_exe)
 faire
  lire l'instruction du pointeur.
  Lire les paramètres.
  Suivant l'instruction faire l'action spécifique.
 fin
fin
```

2. Les différents types d'instruction

Je vais maintenant décrire les différents types d'instruction possibles. Celles-ci sont très peu nombreuses par rapport à un vrai assembleur mais elles sont suffisantes pour les programmes écrits dans notre langage.

2.1. Les instructions. Les instructions se divisent en 3 groupes : les instructions de manipulation de la mémoire, les instructions de piles et les instructions de sauts. Les premières servent à additionner des valeurs, déplacer des valeurs...
Les secondes servent à empiler des valeurs ou à les dépiler.
Les dernières servent à sauter d'une partie d'un programme à une autre suivant les valeurs en mémoire. Voici la liste des instructions possibles ("instr adr1 adr2"). :

- ORG : origine, sauter à l'instruction pointée par adr1. Adr1 est toujours en mode immédiat.
- PUSH :on empile adr1 sur la pile de la machine
- POP : adr1, on dépile sur adr1, 0 si la pile est vide. Adr1 est toujours en mode direct.
- ADD : on incrémente l'adr1 de adr2. Adr1 est en mode direct et Adr2 en mode immédiat.
- SUB : on décrémente l'adr1 de adr2. Adr1 est en mode direct et Adr2 en mode immédiat.
- MOV : adr1 prend la valeur adr2. Adr2 ne peut pas être en mode immédiat.
- MOD : on fait un modulus de adr1 par adr2. Adr1 ne peut être en mode immédiat.
- END : fin de l'exécution
- INT : suivant la valeur de adr1, en mode direct(mais compris en mode immédiat)
  - 1 <=> lire une valeur et la mettre à l'adresse 0
  - 2 <=> écrire sur la sortie standard la valeur de l'adresse 1
  - 3 <=> ouvrir le mode graphique
  - 4 <=> fermer le mode graphique
  - 5 <=> effacer le mode graphique
  - 6 <=> placer un point en (adresse 1,adresse 2) et de couleur adresse 3
- JMP : sauter à l'instructon adr1.
- JZ : sauter à l'instruction adr1 si la valeur à l'adr2 est à zéro.
- JNZ : sauter à l'instruction adr1 si la valeur à l'adr2 est différente de zéro
- JE : si les deux valeurs sont égales, alors la valeur de adr1 est mise à 0

2.2. Exemples de code en C. L'instruction add

```
void add(instruction ins)
debut
 int tmp1,tmp2;
 /* on interdit de faire 4:=A+1 */
 si (ins.type1==1) instr_errone();
 /* On prend les valeurs */
 tmp2=donne_valeur_deux(ins);
 tmp1=donne_valeur_un(ins);
 /* on effectue l'opération, si on passe en dessous de zéro on remet au max */
 si (ins.type1==0)
   si (memoire[ins.adr1]+tmp2<0) memoire[ins.adr1]=INT_MAX;
    else memoire[ins.adr1]+=tmp2;
  else
    si (memoire[memoire[ins.adr1]]+tmp2<0)memoire[memoire[ins.adr1]]=INT_MAX;
```

```
     else memoire[memoire[ins.adr1]]+=tmp2;
fin
```

L'instruction JZ

```
int jz(instruction ins)
debut
 int tmp1,tmp2;
 /* On prend les valeurs */
 tmp2=donne_valeur_deux(ins);
 tmp1=donne_valeur_un(ins);
 /* on effectue oui ou non le changement d'adresse */
 si (tmp1==0)
   si (ins.type2==1) retourne ins.adr2;
     else retourne memoire[ins.adr2];
  else retourne pointeur+1;
  fprintf(stderr,"erreur de jz la");
  exit(EXIT_FAILURE);
fin
```

D'autres actions très simples

- END : fin_exe=vrai ;
- ORG : pointeur=donne_valeur_un(programme[pointeur]) ;
- PUSH :tmp=donne_valeur_un(programme[pointeur]) ;
  pile_exe=Push(pile_exe,tmp) ; pointeur++ ;
- etc...

Maintenant que nous avons vu le fonctionnement de la machine virtuelle et des instructions possibles, nous pouvons commencer à décrire la transformation des programmes LAB, dans notre langage d'assemblage.
Nous supposerons que nous avons une procédure permettant d'insérer une instruction exécutable et sa représentation en langage d'assemblage.

CHAPITRE 3

Génération de code des instructions de base

La génération de code est la partie la plus technique d'un compilateur (même si, ici, le code est pour une machine virtuelle, très proche des PC). Comme le nombre d'instructions possibles est très limité, ces instructions ont été directement mises dans la grammaire. Ainsi, lors de l'analyse syntaxique, quand celle-ci a détecté une de ces instructions, nous pouvons directement générer le code associé. Mais cette génération nécessite une table des symboles, que nous allons maintenant décrire.

1. Table des symboles

La table des symboles de LAB comporte un grand nombre de tableaux :
- Une table des variables globales
- Une table des tableaux globaux
- Une table des paramètres en cours
- Une table des boucles (voir chapitre suivant)
- Une table des hyperboucles (voir chapitre suivant)
- Une table comportant des informations pour chaque fonction

Par souci de simplicité, les tables ne sont pas créées dynamiquement (mais c'est une optimisation possible du compilateur) et sont construites dès le départ de taille prédéfinie par des constantes. Les tables sont toutes des tableaux d'enregistrements que nous allons détailler [1].

1.1. Tables des variables. Une variable est une entité comportant un nom et une adresse. La structure de la table est donc :

```
typedef struct Cellule_var
 {
  char nom_var[max_car];
  int adresse;
 } cellule_var;
```

On peut maintenant créer la table des variables globales, et dans chaque fonction, la table des variables locales.

1.2. Table des tableaux. Un tableau est une entité comportant un nom, une adresse de départ des données et une taille. La structure de la table est donc :

```
typedef struct Cellule_tab
 {
  char nom_tab[max_car];
  int adresse;
  int taille;
 } cellule_tab;
```

On peut maintenant créer la table des tableaux globaux, et dans chaque fonction, la table des tableaux locaux.

[1] Remarque : je ne vais pas décrire les recherches de présence dans les tables car ces recherches sont séquentielles et triviales (sauf un exemple).

1.3. Les paramètres. Les paramètres sont des entités provisoires comportant un nom, une adresse et un type (0 pour variable et 1 pour tableau). Ils font référence à des entités déjà existant. La structure de la table est donc :

```
typedef struct Cellule_param
 {
  char nom_param[max_car];
  int adresse;
  bool type; /* 0 var , 1 tablo */
 } cellule_param;
```

On pourra donc créer les paramètres des fonctions lors de leurs définitions et construire le passage en paramètres des entités dans les corps des fonctions.

1.4. Une fonction. La fonction, est en LAB, l'entité sûrement la plus compliquée et nécessitant le plus d'informations. Une fonction comporte un nom et une adresse de début de code. Elle comporte aussi les tables suivantes : une table des variables locales, une table des tableaux locaux, une table des paramètres. Chaque table a aussi son nombre d'entités ainsi créées (nombre de variables etc...). La structure en C est donc la suivante :

```
typedef struct Cellule_fun
 {
  char nom_fun[max_car]; /* Nom de la fonction */
  cellule_var table_var[max_var]; /* Table des variables de la fonction */
  cellule_tab table_tab[max_tab]; /* Table des tableaux de la fonction */
  cellule_param table_param[max_param]; /* Tables des parametres */
  int nb_param; /* nombre de parametre */
  int nb_var; /* nombre de variables */
  int nb_tab; /* nombre de tableaux */
  int adresse; /* adresse de début de code */
 } cellule_fun;
```

On crée donc une immense table des symboles pour les fonctions.

Remarque : le programme principal pourra ainsi être vu comme une fonction car il a les mêmes propriétés. Il faudra simplement faire attention à ne pas retourner de valeurs mais à simplement quitter le programme.

Remarque : les fonctions prédéfinies de LAB, seront rajoutées dès le début dans la table des symboles. Ainsi, elles seront traitées comme toutes les autres fonctions dans la cohérence des données.

1.5. Avantages et exemples. Liste non exhaustive des avantages
Avec ce système, les avantages sont très nombreux en utilisant des recherches exhaustives :

- On peut facilement tester le nombre de paramètres et leur cohérence avec la définition de la fonction.
- On peut vérifier facilement si une entité a déjà été créée dans les entités locales, globales ou dans les paramètres.
- On peut trouver facilement les fonctions, variables, tableaux inconnus.
- On peut éviter les appels récursifs
- On retrouve facilement les informations nécessaires à chaque entité, car toutes les informations utiles sont sauvegardées dans des tables.

Recherche pour savoir si une variable est déjà globale ou non

```
bool deja_var_global(char *name)
debut
 int i;
 pour(i=0;i<nb_var_global;i++)
  si (strcmp(name,table_var_global[i].nom_var)==0) retourne vrai;
```

```
 retourne faux;
fin
```

2. Les instructions avec variables

Les instructions n'utilisant que des variables sont à la base de LAB. Ce sont elles qui permettent de véritables évolutions dans l'exécution des programmes. Elles sont au nombre de 4 :

- 1) VAR1 :=VAR2 ;
- 2) VAR1 :=NOMBRE ;
- 3) VAR1 :=VAR1+NOMBRE ;
- 4) VAR1 :=VAR1-NOMBRE ;

Notre langage d'assemblage a donc été créé pour que ces instructions ne fassent qu'une instruction machine, vu qu'elles seront les plus utilisées. L'analyse lexicale (avec l'automate d'état et la table des symboles) permet de trouver facilement l'adresse de nos variables. Il est donc aisé de générer le code.

1) Les variables étant simplement des adresses de la mémoire, on utilise l'adressage direct. Nous avons donc le code :

```
MOV VAR1 VAR2
```

2) Maintenant nous utilisons l'adressage immédiat pour représenter notre nombre :

```
MOV VAR1 #NOMBRE
```

3) 4) En utilisant un mélange des deux précédents

```
ADD VAR1 #NOMBRE
SUB VAR1 #NOMBRE
```

On remarque que la génération ressemble plus à un transducteur, vu que la génération est "automatique".

Mais pour les tableaux, cette génération est un peu plus compliquée.

3. Les instructions avec tableaux

En LAB, les tableaux sont cycliques, c'est à dire que l'indiçage est modulo de la taille du tableau. Il faut donc avoir quelque part en mémoire, cette taille. Les manipulations sur les tableaux nécessiteront en permanence cette information.

3.1. Représentation d'un tableau en mémoire. Comme nous l'avons déjà indiqué, un tableau a une taille, c'est à dire un nombre de cases. Quand on déclare "Tab[X]", on permet à l'utilisateur d'avoir des cases de 0 à X compris. Comme nous le verrons au chapitre suivant, on peut passer des tableaux en paramètres. Mais on ne peut pas indiquer directement la taille car on aurait des problèmes de compatibilité et de portabilité de la fonction. J'ai donc, pour les tableaux, utilisé la représentation en mémoire suivante.

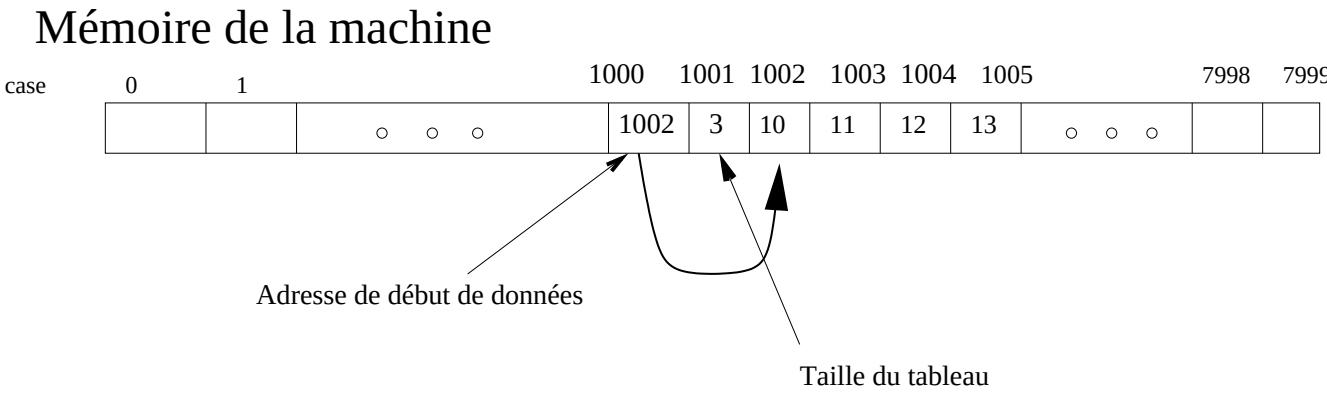

FIG. 1. Représentation d'un tableau Tab[3]

Ainsi il est très aisé de retrouver les informations dans un tableau, grâce à l'adresse de début de données.

3.2. Utilisation. Les instructions utilisant les tableaux sont plus nombreuses mais mais pas assez pour ne pas être mises directement dans la grammaire du langage. Nous utiliserons le nom générique Tab, pour un tableau, VAR pour les variables et NOMBRE pour les entiers naturels.

- 1) Tab[NOMBRE] :=VAR ;
- 2) Tab[NOMBRE] :=NOMBRE ;
- 3) Tab[VAR] :=VAR ;
- 4) Tab[VAR] :=NOMBRE ;
- 5) VAR :=Tab[VAR] ;
- 6) VAR :=Tab[NOMBRE] ;

On le voit ici, les tableaux servent essentiellement à stocker des données.
Pour y accéder, il faut d'abord faire un modulus de l'indice avec la taille du tableau. Puis ajouter à cette nouvelle valeur, l'adresse de début des données. On obtient ainsi, l'adresse de la case mémoire désirée. Pour cela nous utiliserons la case mémoire 0 réservée à la machine virtuelle. Nous n'allons pas détailler toutes les instructions à générer mais utiliser simplement des exemples significatifs. ADR est l'adresse du tableau qui contient l'adresse de début de données. Donc (ADR+1) représente case contenant la taille du tableau.

```
Instruction de type 1)
MOV 0 #NOMBRE
MOD 0 (ADR+1)
ADD 0 ADR
MOV @0 #NOMBRE
```

```
Instruction de type 4)
MOV 0 VAR
MOD 0 (ADR+1)
ADD 0 ADR
MOV @0 #NOMBRE
```

```
Instruction de type 5) VAR1:=Tab[VAR2]
MOV 0 VAR2
MOD 0 (ADR+1)
ADD 0 ADR
MOV VAR1 @0
```

Apparaît clairement le rôle extrêmement important de l'adressage indirect. Il peut être considéré comme un pointeur (représentation par flèche) et comme on n'utilise que des entiers, le pointeur n'a pas besoin de type. Ce mode d'adressage est très utilisé en assembleur mais il doit être manipulé avec soin si on veut ne pas pointer en dehors de la mémoire.

Maintenant que nous avons la génération des instructions de base (les petites briques du langage), nous allons pouvoir passer à la génération d'une autre partie très importante de LAB : les fonctions.

CHAPITRE 4

Génération de code des appels de fonction

En effet, les fonctions permettent un découpage des programmes et l'utilisation d'outils génériques qui peuvent servir à tout moment (voir manuel d'utilisation).

Mais les fonctions ont besoin d'arguments pour être utilisées. Se posent alors deux questions : comment créer les paramètres ? Comment ensuite les utiliser ? A la première question, il suffit de faire comme à la création d'entité. On ajoute en mémoire les entités utilisées. A la seconde, nous allons passer les arguments par piles.

1. Définition et appel de fonction

Beaucoup de fonctions utilisent des arguments. Ces arguments vont être manipulés et avoir de l'influence sur l'exécution du programme. En LAB, les paramètres sont toujours passés "en paramètres" (comme désignés dans les autres langages de programmation).

1.1. Définition. Pour cela, pour chaque argument donné, on utilise une case mémoire. Ceci peut être fait grâce à l'automate d'état (voir précédemment) qui permet de savoir si on est dans les arguments de la fonction ou dans la création locale. Il faudra vérifier que les entités ne sont pas des entités globales car cela peut porter à confusion. On réserve donc de la place dans la mémoire pour ces entités. Pour récupérer les valeurs données lors de l'exécution du programme, il faudra dépiler les valeurs. Exemples :

```
entier fun(A,B,C)
debut
...
fin
A est réservé à la case mémoire 10, B à la case 11 et C à la case 12
<==>
POP 12
POP 11
POP 10
```

Après on continue normalement la génération de code.

1.2. Appel de fonction. Un appel de fonction se fait en plusieurs étapes. Il ne peut se faire que quand on a lu tous les arguments à mettre en paramètres (c'est à dire après ")"). On utilise donc encore l'automate d'état qui a mis chaque entité lue dans une table d'arguments temporaire. Algo grossier pour (VAR :=fun(A,B,C,...)) :

- Vérifier si on ne fait pas un appel récursif
- Vérifier que la fonction appelée existe bien par un parcours de la table des symboles
- Vérifier le nombre d'arguments et leur cohérence par un parcours des arguments de la fonction
- Si c'est une fonction système alors générer un code spécial
- Empiler l'adresse de retour d'instruction c'est à l'adresse de l'instruction + nombre d'arguments + 1
- Empiler les arguments (A,B,C...)
- écrire un saut (jmp) à l'adresse de début de code de la fonction appelée
- écrire un "pop VAR" pour récupérer la valeur de retour de la fonction appelée.

On voit ici l'intérêt des instructions de sauts. Exemple, appel de fonction : A :=F(A,B), ou A est à l'adresse 10, B en 11 et la fonction à son début de code en 1

```
26:push #30
27:push 10
28:push 11
29:jmp #1
30:pop 11
```

On voit qu'un appel de fonctions génère en fait beaucoup d'instructions machine mais les fonctions sont tellement utiles... On a donc le schéma suivant :

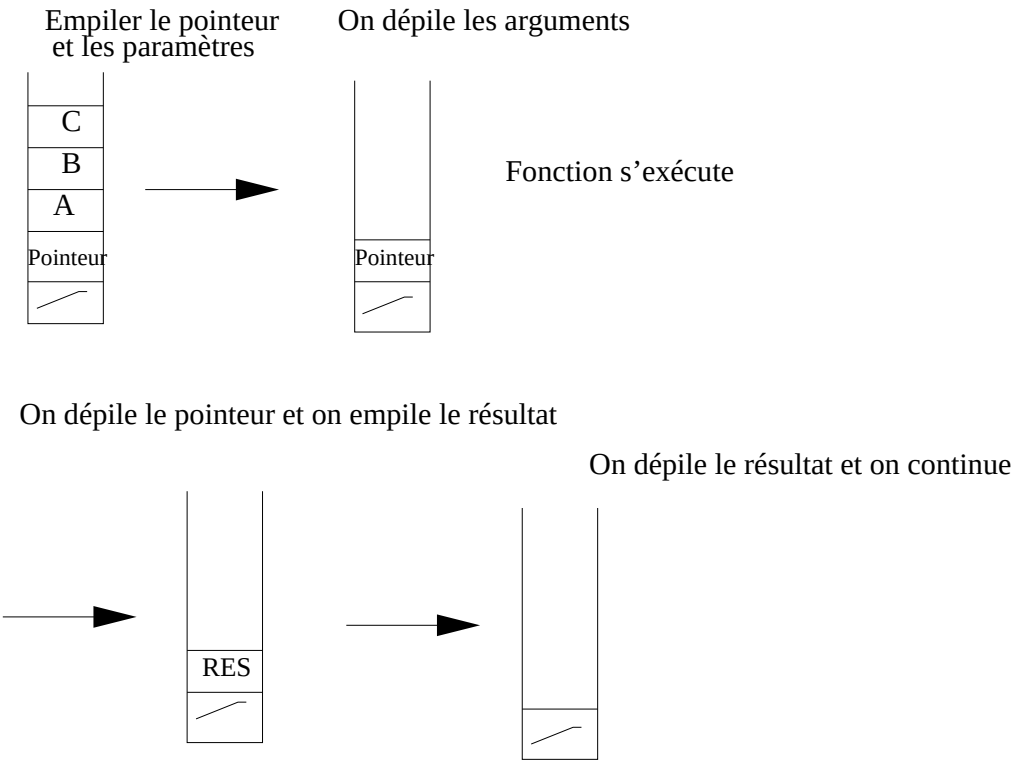

FIG. 1. Exemple d'appel de fonction

1.3. Fonctions système. Pour les fonctions prédéfinies en LAB (écrire, pixel, lire, etc...), il faut générer un code spécial. Celui-ci est différent pour chaque instruction. Je donne le code en C comme exemple. Val1 est la variable qui recevra la valeur de retour. On utilise aussi les adresses réservées au système.

lire

```
add_instr(INT,1,0,0,0);
add_instr(MOV,val1,0,0,0);
```

ecrire

```
add_instr(MOV,1,0,parametre_en_cour[0].adresse,0);
add_instr(INT,2,0,0,0);
add_instr(MOV,val1,0,0,1);
```

ouvregraph

```
add_instr(INT,3,0,0,0);
add_instr(MOV,val1,0,0,1);
idem, pour fermegraph et effacer.
```

pixel

```
add_instr(MOV,1,0,parametre_en_cour[0].adresse,0);
add_instr(MOV,2,0,parametre_en_cour[1].adresse,0);
add_instr(MOV,3,0,parametre_en_cour[2].adresse,0);
add_instr(INT,6,0,0,0);
add_instr(MOV,val1,0,0,1);
```

1.4. Changement de fonction. Au fur et à mesure de la génération de code, on change de fonction. Il faut donc mettre la table des symboles à jour. Ceci montre comment se servir de cette table et comment le tout est codé.

```
void change_fun()
debut
 int i;
/* ******************* Si on a atteint le nombre maximal de fonctions ********* */
  si (nb_fun_lu==max_fun)
   debut
     erreur_ligne();
     fprintf(stderr,"Trop de fonctions demandees :%s, plus de mémoire\n",nom_fun_en_cours);
     exit(0);
   fin
/************** on teste si la fonction n'a pas déjà été définies ******** */
  pour(i=0;i<nb_fun_lu;i++)
    si (strcmp(table[i].nom_fun,nom_fun_en_cours)==0)
     debut
      erreur_ligne();
      fprintf(stderr,"Fonction déjà définie plus haut\n");
      exit(0);
     fin
/* ************** On change de nom de fonction courante ************* */
  strcpy(table[nb_fun_lu].nom_fun,nom_fun_en_cours);
/* ************* On met à jour l'adresse de départ de la fonction ******** */
  table[nb_fun_lu].adresse=ligne_asm;
/* ************ On augmente le nombre de fonctions lues ****************** */
  nb_fun_lu++;
fin
```

Erreur_ligne indique la ligne de l'erreur.

Pour le programme principal, on utilisera une fonction presque identique (voir code).

2. L'instruction "retourner" et fin de fonction

Comme nous pouvions nous en douter, une fonction retourne une valeur calculée. Ceci peut être effectué par l'instruction (qui en fait une fonction comme les autres) "retourner" (voir manuel d'utilisation). Cette "fonction" arrête donc l'exécution de la fonction et doit permettre de retourner une valeur. Pour cela, il suffit de dépiler pour retrouver l'ancienne valeur du pointeur d'exécution (revenir au code qui avait fait appel à la fonction) et empiler le résultat calculé. On utilisera une adresse réservée au système.

2.1. Code en C.

```
add_instr(MOV,val1,0,0,1);
        add_instr(POP,0,0,0,0);
        add_instr(PUSH,parametre_en_cour[0].adresse,0,0,0);
        add_instr(JMP,0,0,0,0);
```

2.2. La fin de fonction. Il est dit dans le manuel d'utilisation que LAB assure toujours un retour de valeur. Donc si l'utilisateur omet de mettre un "retourner" dans sa fonction, LAB retourne une valeur nulle. Pour cela il suffit de de dépiler le pointeur d'instruction, d'empiler 0 et de faire un saut. Ceci est fait automatiquement à chaque fin de fonction car il est impossible de savoir si le code permet de renvoyer une valeur.

3. Le problème des tableaux

Depuis le début de ce chapitre, les paramètres sont de simples variables. Qu'en est-il des tableaux ? Le passage en paramètre de tableaux semblent poser un problème. Nous rappelons qu'un tableau est défini par son nom, sa taille et son ADRESSE de DEBUT de code qui est en fait un pointeur sur les données.

En LAB les tableaux modifiés par une fonction sont modifiés pour tout le programme. Nous allons donc utiliser cette propriété en créant un nouveau pointeur sur les données. Ce pointeur permettra d'accéder aux données du tableau sans toucher aux autres pointeurs. Pour accéder à la taille du tableau, on sait que la taille est placée avant les données, elle est donc encore facile d'accès pour la génération de code car il suffira de l'empiler.
Ce pointeur servira pour toute la fonction (et même pour les appels de fonction). On évite ainsi la copie de données et le passage en paramètre d'un tableau ne coûte pas plus qu'un passage en paramètre d'une variable. Schéma :

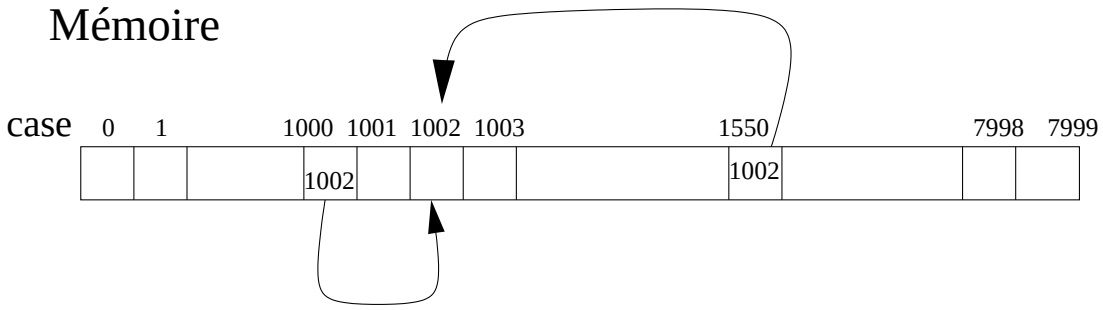

FIG. 2. Passage en paramètre d'un tableau

Le passage en paramètre de tableaux n'est donc pas un problème.

4. Exemple d'empilement des arguments

```
void Poper(char *name_fun)
debut
 int j,i; j=nb_fun_lu-1;
  /* On pop en sens inverse des push donc de la fin vers le début */
  pour(i=table[j].nb_param-1;i>=0;i--)
   debut
    /* Si variable un simple pop */
    si (table[j].table_param[i].type==0)
     debut
      add_instr(POP,table[j].table_param[i].adresse,0,0,0);
      fin
    else
    /* Si tableau, on pop la taille puis l'adresse */
     debut
       add_instr(POP,table[j].table_param[i].adresse+1,0,0,0);
       add_instr(POP,table[j].table_param[i].adresse,0,0,0);
     fin fin fin
```

CHAPITRE 5

Génération de code des boucles

Les boucles sont les premières instructions itératives de LAB. Elle permettent d'effectuer un nombre prédéfini de fois une suite d'instructions. Pour cela, elles utilisent chacune un compteur qui est décrémenté à chaque appel.

1. Création d'une boucle

Il faut donc créer un compteur, c'est à dire utiliser une nouvelle case mémoire. Ensuite le compteur est initialisé à la valeur souhaitée et il faut mettre des sauts, l'un au début qui teste la nullité du compteur et un autre à la fin qui retourne toujours sur le test de nullité.

1.1. Algorithme.

```
Créer un compteur
Initialiser ce compteur avec la valeur voulue
Mettre une instruction de saut
Continuer la génération
Mettre une autre instruction de saut.
```

Exemple

```
boucle A fois
 P
fin
P'
On supose A est à la cases 10. Le compteur à la case 12
<=>
23:mov 12 10
24:jz 12 #28 <------------|  -----------|
25:le code de P           |             |
26:sub 12 #1              |             |
27:jmp #24 ---------------|             |
28:le code de P' <----------------------|
```

Le code de P sera donc bien effectué A fois. Puis on effectuera le code P'.

1.2. Où sauter ? Le problème qui se pose est le suivant : quand on insère l'instruction "jz" on ne sait pas où est la fin de la boucle (en nombre d'instructions machine)

La solution est d'écrire temporairement dans un tableau. On peut donc "repasser" sur le code pour mettre à jour le "jz" et puis écrire le tableau sur le disque. Mais se pose un nouveau problème : comment faire pour les boucles imbriquées ? La solution est donc une table des "boucles" pour mémoriser les informations sur les "boucles".

1.3. La fonction "sortir". Cette fonction permet de "sortir" d'une boucle (voir manuel d'utilisation). Elle prend en paramètre. Si ce paramètre est différent de 0 alors on "sort" de la boucle. Sortir de la boucle correspond donc à sauter à la fin de la boucle. C'est comme précédemment, on ne peut connaître la valeur de ce saut qu'en ayant effectué le reste de la compilation (dans la boucle où se trouve la fonction). Il faut donc mémoriser les "sortir" de la boucle pour de même repasser le code et mettre à jour les "sortir".

2. Une table pour calculer les sauts

Cette table devra contenir les informations suivantes :
- l'adresse de début de boucle
- adresse du saut de fin de boucle
- le nombre de "sortir" de la boucle
- les adresses des "sortir" de la boucle

Ceci donne en C

```
typedef struct Cellule_boucle
 {
  int adr_jz; /* adresse de début de boucle, le JZ voir rapport */
  int val_jz; /* adresse du compteur de boucle */
  int adr_jmp; /* adresse du JMP à la fin de la boucle */
  int nb_sort; /* nombre de "sortir" croisés dans la boucle */
  int sort[max_sort]; /* table des "sortir" */
 } cellule_boucle;
cellule_boucle pile_boucle[max_boucle];
int nb_boucle; /* nombre de boucles */
```

Ainsi, on peut retenir toutes les informations nécessaires sur les boucles. A la fin d'une, on repasse le code, en remettant bien à jour les sauts.

La plus simple des solutions, et sûrement la meilleure, est de mémoriser toutes les instructions machine pour la fonction en cours. On n'écrira sur disque qu'à la fin de la fonction (on parcourt la table en écrivant les instructions ainsi mémorisées).

2.1. Algo.

```
Mettre le sub
Mettre le jump
Mettre le saut de début boucle à jour
pour tous les "sortir" mettre la valeur de saut à jour
```

Problème de décalage

Il faut faire attention au fait que, dans notre tableau les adresses sont relatives, et qu'il faut additionner avec la valeur du total des instructions générées.

Code en C

```
int fin_boucle(void)
debut
 int i;
 /* on decremente le compteur et on saute */
 add_instr(SUB,pile_boucle[nb_boucle-1].val_jz,0,1,1);
 add_instr(JMP,pile_boucle[nb_boucle-1].adr_jz,1,0,0);
 /* on met l'instruction de saut de debit de boucle a jour (JZ) */
 bloc_instr[pile_boucle[nb_boucle-1].adr_jz-ligne_asm+nb_instr].adr2=ligne_asm;
 bloc_instr[pile_boucle[nb_boucle-1].adr_jz-ligne_asm+nb_instr].type2=1;
 bloc_instr[pile_boucle[nb_boucle-1].adr_jz-ligne_asm+nb_instr].adr1=pile_boucle[nb_boucle-1].v
 bloc_instr[pile_boucle[nb_boucle-1].adr_jz-ligne_asm+nb_instr].type1=0;
 /* On remet à jour tous les "sortir" */
 for(i=0;i<pile_boucle[nb_boucle-1].nb_sort;i++)
 debut
   bloc_instr[pile_boucle[nb_boucle-1].sort[i]-ligne_asm+nb_instr].adr2=ligne_asm;
   bloc_instr[pile_boucle[nb_boucle-1].sort[i]-ligne_asm+nb_instr].type2=1;
 fin
 nb_boucle--;
fin
```

3. Défauts

Le principal défaut du compilateur vient du fait que chaque variable et tableau utilise une ou plusieurs cases mémoire. Ainsi, si on n'utilise pas pour le moment la fonction, elle prend de la place inutilement dans la mémoire. Par contre, on évite ainsi le chargement dans la mémoire de la fonction.

De même, chaque boucle utilise un compteur, même si la boucle ne sert pas pour le moment. L'amélioration possible serait une mémoire dynamique avec chargement de fonctions etc... Mais ceci pourrait être fait en transformant mon langage d'assemblage en langage machine spécialisé (pour PC,RISK,SUN etc...).

4. Schéma des interactions des fichiers

Pour comprendre, le fonctionnement général du compilateur et de la machine virtuelle, j'expose les interactions entre ces différentes parties.

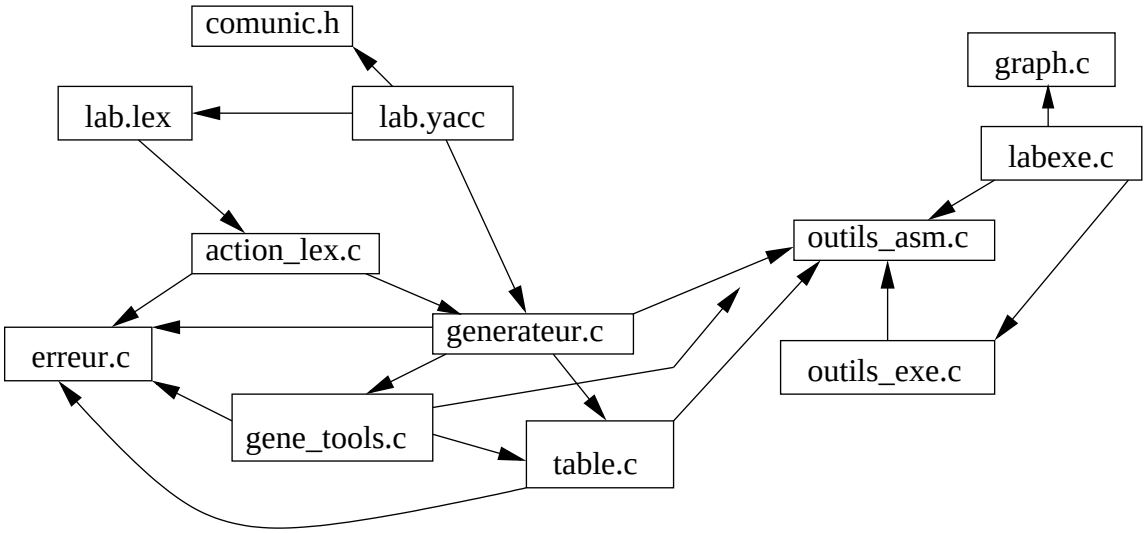

FIG. 1. Les interactions entre les différentes parties du compilateur

De plus, chaque partie utilise une partie nommée "outils", où se situent des outils de programmation (macro...) et la partie des variables qui sont utilisées par toutes les parties (tables des symboles, nombre de lignes...).

CHAPITRE 6

Images de démonstration

Je présente, dans ce chapitre, des captures d'écran de compilation et d'exécution de certains programmes.

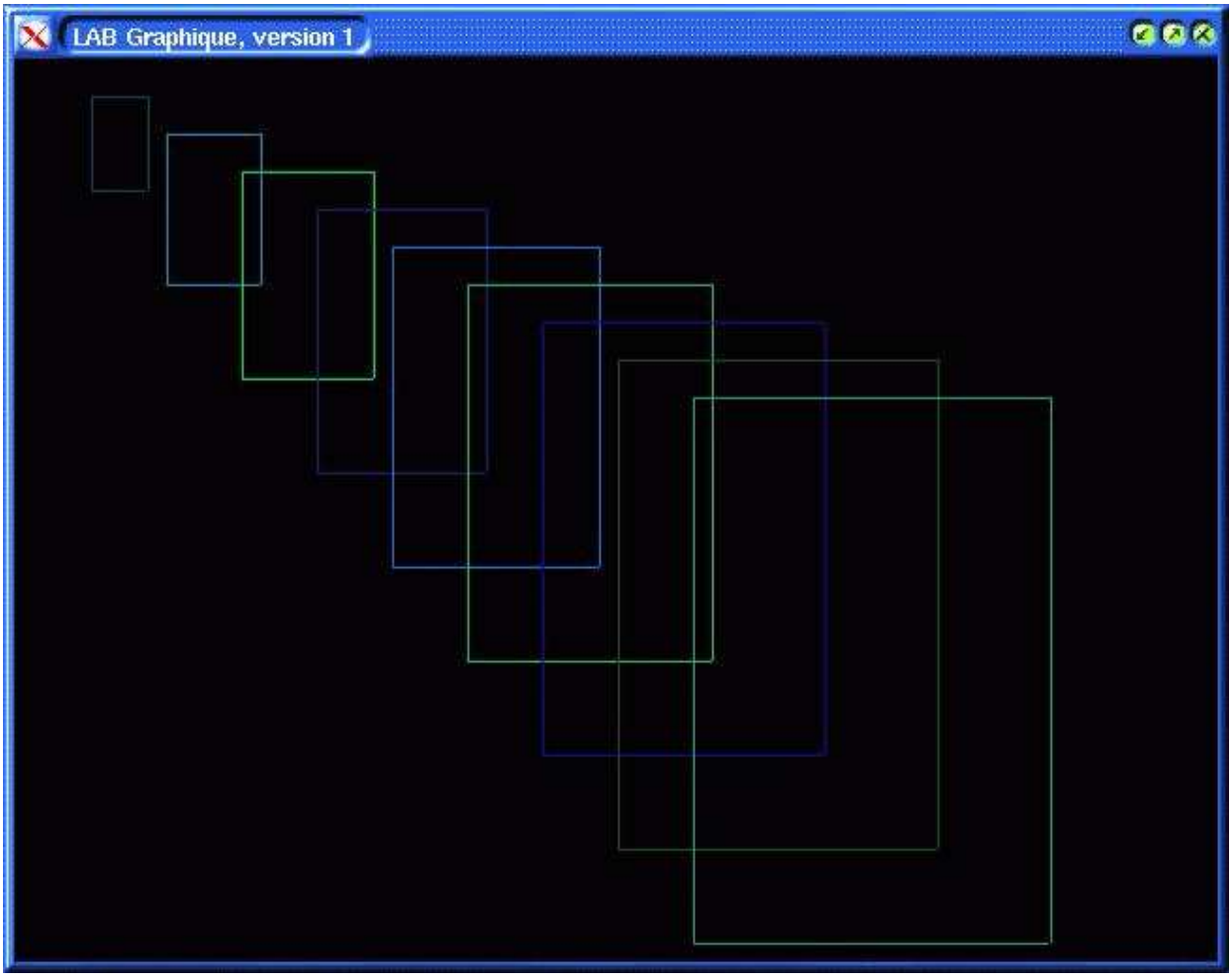

FIG. 1. Le mode graphique, pour des applications ludiques

Et le code permettant d'obtenir le desssin :

```
/*
 Il faut utiliser le modules maths, pour l'addition
  et la variable TMP
*/
#creation d'une ligne horizontale
entier lignehori(X,Y,LEN,C)
creation:X1,Y1
debut
```

```
 X1:=X;
 Y1:=Y;
 boucle LEN fois
   TMP:=pixel(X1,Y1,C);
   X1:=X1+1;
 fin
fin

#création d'une ligne verticale
entier lignever(X,Y,LEN,C)
creation:X1,Y1
debut
 X1:=X;
 Y1:=Y;
 boucle LEN fois
   TMP:=pixel(X1,Y1,C);
   Y1:=Y1+1;
 fin
fin

/* création d'un rectangle
  on utilise les deux fonctions précédentes pour dessiner
   les rectangles */
entier rectangle(X,Y,LEN1,LEN2,C)
creation:Z1,Z2
debut
 TMP:=lignehori(X,Y,LEN1,C);
 TMP:=lignever(X,Y,LEN2,C);
 Z1:=X;
 X:=add(X,LEN1);
 TMP:=lignever(X,Y,LEN2,C);
 X:=Z1;
 Y:=add(Y,LEN2);
 TMP:=lignehori(X,Y,LEN1,C);
fin

programme(N)
creation:X,Y,L1,L2,C
debut
 TMP:=ouvregraph();
 X:=1;
 Y:=Y;
 L1:=10;
 C:=1;
 L2:=20;
 boucle N fois
   TMP:=rectangle(X,Y,L1,L2,C);
   C:=C+300;
   X:=X+40;
   Y:=Y+20;
   L1:=L1+20;
   L2:=L2+30;
   fin
 TMP:=lire(); #Lire une valeur pour pouvoir visualiser le résultat
 TMP:=fermegraph();
fin
```

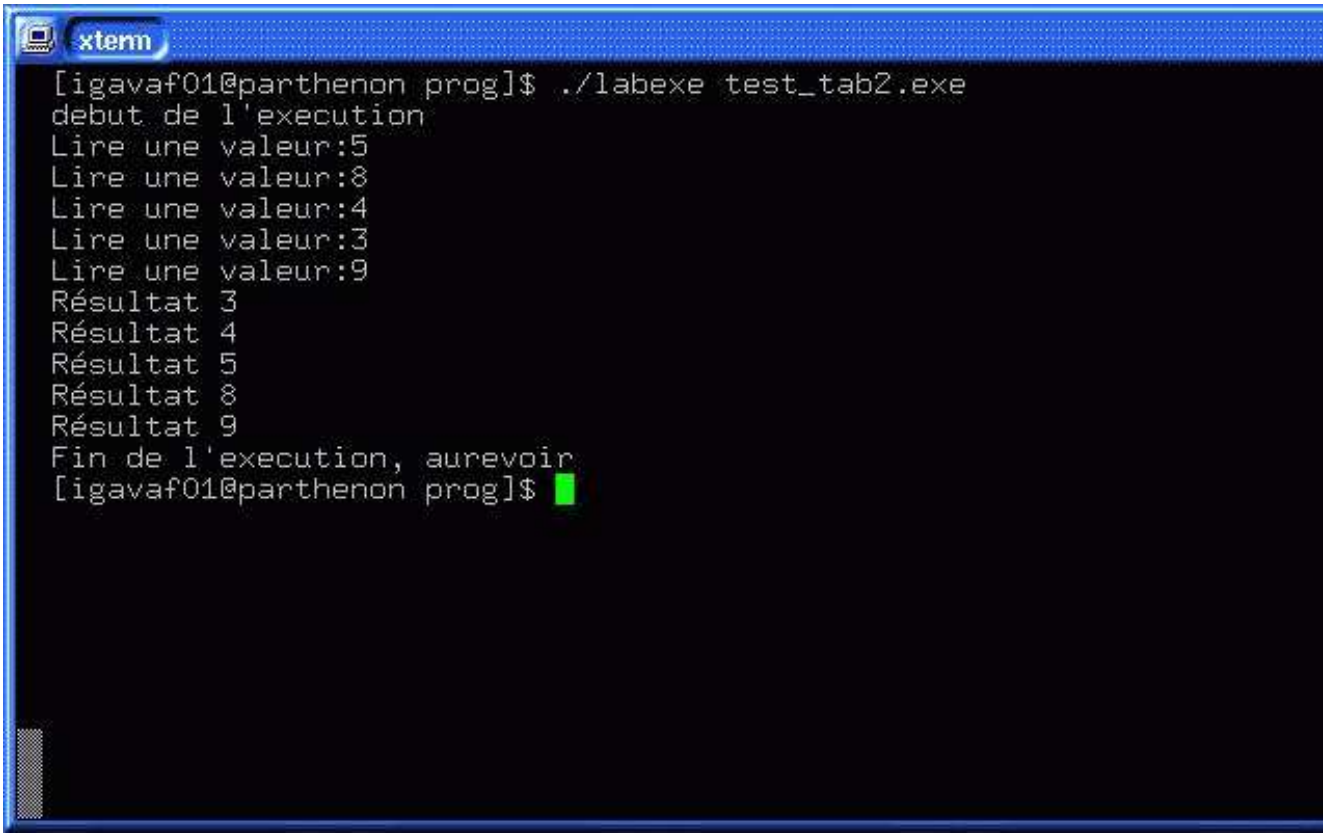

FIG. 2. Exemple d'un tri à bulles dans un tableau

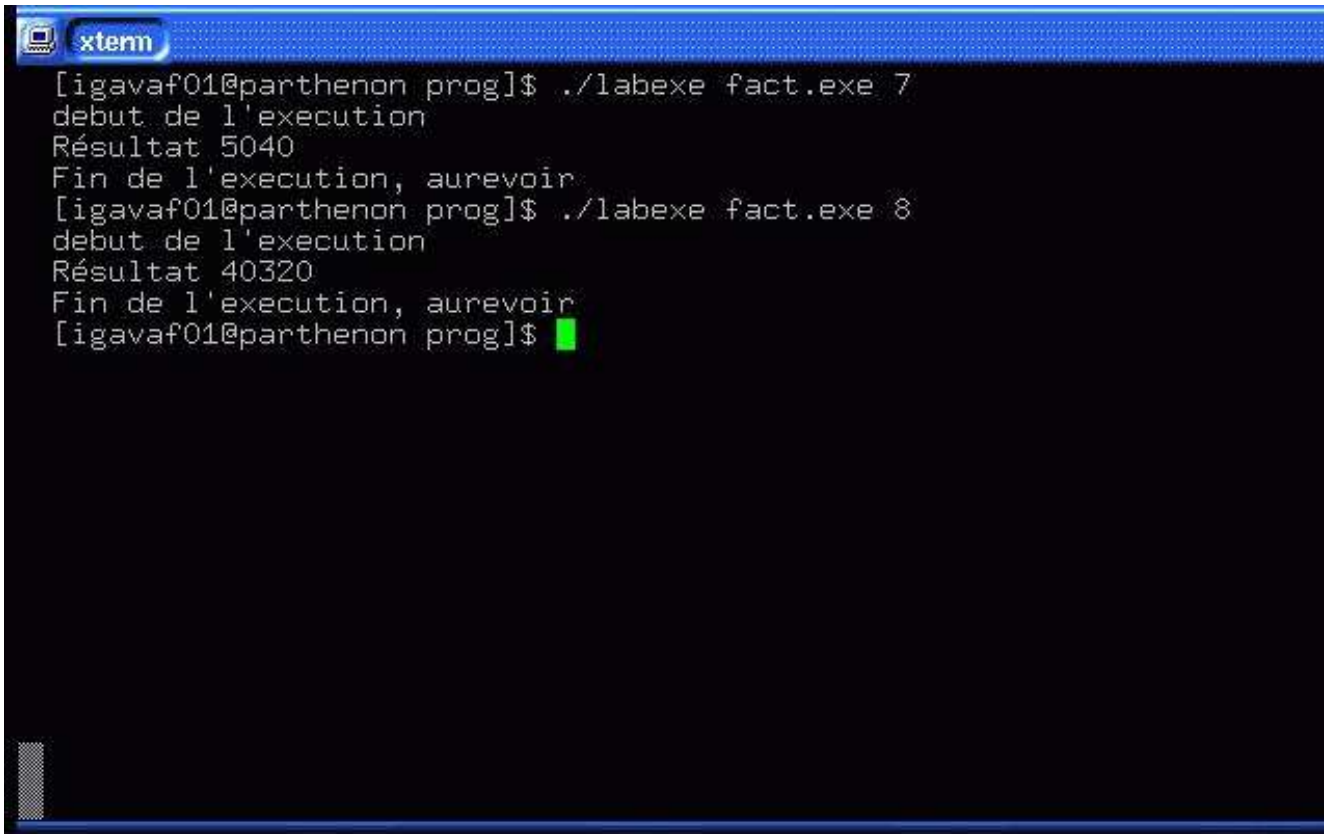

FIG. 3. La factorielle en simple boucle

Conclusion

Ici se termine la partie sur le compilateur LAB. LAB n'est pas un compilteur extrêmement puissant (aucune optimisation de code) mais il est suffisamment rapide pour être utilisé à des fins pédagogiques.

Dans la partie suivante, tous les algorithmes ne seront pas écrits en LAB, mais les implémentations machine le seront.

Deuxième partie

Les algorithmes de base

Introduction

Dans la dernière partie, nous avons décrit un petit compilateur pour un langage de programmation très simple à utiliser. Ce langage peut paraître trop peu puissant pour être utilisé et construire des algorithmes, même les plus triviaux. Nous allons nous efforcer de montrer, dans cette nouvelle partie qu'en fait un tel langage est suffisant pour toute une partie de l'algorithme de base et qu'il permet de garder la complexité par rapport aux langages classiques

Dans cette partie, Il faut bien distinguer les fonctions (écrites de manière mathématique), les algorithmes et les programmes (ou procédures). En fait, ces dernières ne sont que la programmation (dans le langage de la première partie) des algorithmes.

CHAPITRE 7

Fonctions sur les entiers

1. Définitions de base

1.1. Définition des fonctions primitives récursives. Soit $\{N^k \to N | k \geq 0\}$, l'ensemble des fonctions dont les arguments (en nombres quelconques) et la valeur sont des nombres naturels. Nous allons définir un sous-ensemble de ces fonctions que nous appèlerons *fonctions primitives récursives*. Celles-ci sont définies à partir de fonctions de base, d'une règle de composition et d'une règle de récursion.

DÉFINITION 1. *Les fonctions primitives récursives de base sont les suivantes :*

(1) *La fonction*
$$0()$$
est la fonction zéro. *Elle n'a pas d'argument et a toujours la valeur 0.*

(2) *Les fonctions*
$$\pi_i^k(n_1, \cdots, n_k)$$
($1 \geq k$ et $1 \geq i \geq k$) sont les fonctions de projection. La fonction π_i^k a comme valeur le i-ième argument parmi k ($\pi_i^k(n_1, \cdots, n_k) = n_i$).

(3) *La fonction*
$$\sigma(n)$$
est la fonction successeur. *Elle est définie par $\sigma(n) = n + 1$.*

DÉFINITION 2. *Soient g une fonction à l arguments et $h_1, \cdots, h_l$ des fonctions à k arguments. Si nous dénotons $n_1, \cdots, n_k$ par $\overline{n}$, alors la* composition *de g et de $h_1, \cdots, h_l$ est la fonction dans $N^k \to N$ définie par*
$$f(\overline{n}) = g(h_1(\overline{n}), \cdots, h_l(\overline{n}))$$
.

DÉFINITION 3. *Soient g une fonction à k arguments et h une fonction à k+2 arguments. Alors, la fonction f à k+1 arguments telle que :*
- $f(\overline{n}, 0) = g(\overline{n})$
- $f(\overline{n}, m+1) = h(\overline{n}, m, f(\overline{n}, m))$

est la fonction définie à partir de g et h par récursion primitive .

Remarquons que si les fonctions g et h utilisées pour définir une fonction f par récursion primitive sont calculables par une procédure effective, alors f est aussi calculable.

DÉFINITION 4. *Les fonctions primitive récursives sont :*

(1) *Les fonctions primitives récursives de base.*

(2) *Toutes les fonctions obtenues à partir des fonctions primitives récursives de base par un nombre quelconque d'applications de la composition et la récursion primitive.*

1.2. Théorème. Le langage *Loop* a été introduit par Meyer et Ritchie dans les années 70. Les variables en Loop prennent leurs valeurs dans N.

DÉFINITION 5. *Le langage Loop a trois types d'instructions de base*

(1) *X :=0 ; La valeur 0 est assignée à la variable X*

(2) *X :=Y ; La valeur de la variable Y est assignée à la variable X*

(3) *X :=X+1 ; incrémenter d'un la valeur de la variable X*

En plus de ces instructions de base, nous ajoutons l'instruction

```
loop X
 un bloc d'instructions:P
end
```

Dont la sémantique est simple : on exécute X fois le bloc P.
Les programmes *Loop* sont de la forme

$$input X_1, \cdots, X_m$$
$$< \text{des instructions} >$$
$$output X_n, \cdots, X_p$$

THÉORÈME 1. *Les fonctions primitives sont strictement équivalentes aux programmes Loop*

Ainsi toute fonction primitive récursive peut s'écrire sous la forme d'un programme Loop et surtout tout programme *Loop* est équivalent à une fonction primitive récursive.
Démonstration : voir article [10].

THÉORÈME 2. *Toutes les fonctions primitives terminent.*

Donc, par équivalence, les programmes *Loop* terminent. le langage *Loop* n'est donc pas Turing-complet.

Nous allons maintenant, introduire les fonctions primitives récursives les plus classiques avec le langage de programmation décrit dans la première partie. Ce langage est une extention pour programmeur du langage *Loop* (ajoût de procédures, et de quelques autres instructions...).

2. Fonctions de base

2.1. Les nombres. Toutes les fonctions constantes $J() = j$ sont primitives récursives. En effet, chacune se définit par une suite de compositions.

$$J() = \overbrace{\sigma(\sigma(\cdots\sigma(}^{j \text{ fois}} 0())))$$

Un algorithme en *Loop*, donnerait

```
X:=0;
X:=X+1;
...       j fois
X:=X+1;
```

Dans notre langage, nous utiliserons X :=j (ou j est un nombre>=0).

2.2. L'addition. L'addition arithmétique de N1 et N2 est aussi primitive récursive.
On peut l'écrire avec le schéma de récurrence suivant :

$$\begin{array}{lcl} add(N1, 0) & = & \pi_1^1(N1) \\ add(N1, N2+1) & = & \sigma(\pi_3^3(N1, N2, add(N1, N2))) \end{array}$$

ce qui peut se simplifier en

$$\begin{array}{lcl} add(N1, 0) & = & N1 \\ add(N1, N2+1) & = & \sigma(add(N1, N2)) \end{array}$$

Dans notre langage, nous pouvons utiliser :

```
entier add(X,Y)
creation:Z
 debut
   Z:=X;
   boucle Y fois Z:=Z+1; fin
   TMP:=retourner(Z);
 fin
```

2.3. La multiplication simple. La fonction produit est primitive récursive puisqu'elle se définit par récursion primitive à partir de l'addition.

$$\begin{array}{lcl} N \times 0 & = & 0 \\ N \times (M+1) & = & N + (N \times M) \end{array}$$

Dans notre langage, nous pouvons écrire :

```
entier mult(N,M)
creation:Z
debut
 Z:=0;
 boucle N fois
  boucle M fois
     Z:=Z+1;
   fin
 fin
 TMP:=retourner(Z);
fin
```

2.4. L'exponentielle simple. Similairement, la fonction puissance (n^m) se définit par récursion primitive à partir de la multiplication

$$\begin{array}{lcl} N^0 & = & 1 \\ N^{M+1} & = & N \times N^M \end{array}$$

On peut donc la voir comme une suite de multiplications.Voici le programme :

```
entier puissance(N,M)
creation:U
debut
U:=1;
boucle M fois
 U:=mult(M,U);
fin
TMP:=retourner(U);
fin
```

Dans cette méthode, on construit n puis n*n puis n*n*n etc... donc on a une variable (celle de la multiplication) qui va de 0 à n puis de 0 à n*n etc... Il est possible d'écrire une procédure qui fasse la construction de 1 à N^M de manière directe, c'est à dire sans passer par des intermédiaires. Cette construction est un peu plus délicate.

```
entier expo(N,M)
creation:Z
debut
  Z:=1;
  N:=N-1;
  boucle M fois
   boucle Z fois
     boucle N fois Z:=Z+1;
      fin
   fin
  fin
 TMP:=retourner(Z);
fin
```

Nous verrons dans la prochaine partie comment faire pour avoir une processus de contruction "plus proche" de la définition mathématique de l'exponentiation.

3. Fonctions de base plus complexes

3.1. La fonction prédécesseur. Comme nous travaillons sur les entiers naturels, le prédécesseur de 0 est 0.

$$\begin{array}{lcl} pred(0) & = & 0 \\ pred(M+1) & = & M \end{array}$$

Hélas, en *Loop*, on ne peut pas donner le résultat aussi vite. Il faut comme en Lambda-calcul, construire le prédécesseur à partir de 0. Programme :

```
entier pred(M)
creation:Z
debut
 Z:=0;
 boucle M fois
  Z:=Z+1;
  fin
 TMP:=retourner(Z);
fin
```

L'algorithme n'a pas une bonne complexité car il nécessite M additions pour obtenir le prédécesseur. Cette perte de complexité peut se répercuter sur d'autres algorithmes utilisant le prédécesseur.

LEMME 1. *(4) Pour garder une bonne complexité, il est nécessaire d'utiliser une quatrième instruction : X :=X-1 ;*

Justification : se verra sur d'autres algorithmes.

Les complexités se mesurent sur le nombre d'instructions (2), (3) et (4)

3.2. La différence. La fonction différence est définie suivant une convention similaire à celle utilisée pour la fonction prédécesseur.

$$\begin{array}{lcl} n-0 & = & n \\ n-(m+1) & = & pred(n-m) \end{array}$$

le programme va utiliser le schéma de récursion précédent, en enlevant M fois 1 à N.

```
entier diff(N,M)
creation:
debut
 boucle M fois N:=N-1; fin
 TMP:=retourner(N);
fin
```

On voit ici, que la complexité ne serait pas bonne sans cette nouvelle instruction, car on aurait une complexité

$$\begin{array}{lcl} Complexite & = & O(N)+O(N-1)+O(N-2)+\cdots+O(1) \\ ie & = & O(\frac{N\times(N-1)}{2}) \\ ie & \sim & O(N^2) \end{array}$$

alors que l'algo est linéaire. Cette instruction est donc bien nécessaire pour garder une bonne complexité. Nous verrons par la suite, que dans la transformation d'algorithmes avec la boucle While en algorithmes en Loop, cette instruction permet de garder la complexité.

3.3. La factorielle. Elle est définie par la formule suivante :

$$N! = \overbrace{1\times 2\times 3\times 4\cdots(N-1)\times N}^{N \text{ fois}}$$

La fonction factorielle (n!) est elle aussi primitive récursive.

$$\begin{array}{lcl} 0! & = & 1 \\ (N+1)! & = & (N+1)\times N! \end{array}$$

Comme pour l'exponentiation, il n'est pas évident de construire l'algorithme sur ce schéma de récurrence. Programme :

```
entier fact_lent(N)
creation:RES,A
debut
 RES:=1;
 A:=1;
 boucle N fois
  RES:=mult(RES,A);
  A:=A+1;
 fin
 TMP:=retourner(F);
fin
```

ou plus rapide tout en boucle

```
entier fact(N)
creation:RES,A,X
debut
 RES:=1;
 A:=0;
 X:=1;
 boucle N fois
   boucle A fois
      boucle RES fois
        X:=X+1;
      fin
    fin
   A:=A+1;
   RES:=X;
  fin
 TMP:=retourner(RES);
fin
```

la deuxième version, ne faisant pas appel à la procédure de multiplication est plus rapide car on n'a pas l'appel de la procédure, de plus on a pu "ruser" en commençant l'incrémentation à 0 et en ne re-initialisant pas le compteur à 0 (comme il est fait avec la procédure de multiplication, voir l'exponentiation). La complexité est $N!$, dans les deux cas, mais en machine la deuxième version va environ 25% plus vite.

3.4. Suites. Somme des N premiers nombres

La somme des n+1 premiers entiers naturels s'écrit mathématiquement :

$$S = \sum_{i=0}^{i=n} i$$

Mais elle peut aussi s'écrire sous la forme (primitive récursive) suivante

$$\begin{aligned} S(0) &= 0 \\ S(n) &= S(n-1) + n \end{aligned}$$

On peut donc facilement construire la procédure à partir de ce schéma (qui ressemble à celui de la factorielle).

```
entier somme(N)
creation:I,S
debut
 I:=0;
 S:=0;
 boucle N fois
   I:=I+1;
   S:=add(S,I);
   TMP:=ecrire(S);
  fin
TMP:=retourner(S);
fin
```

On peut remarquer que la complexité est polynomiale.

Suite géométrique

La suite géométrique est aussi, une suite classique, qui s'écrit :

$$S = \sum_{i=0}^{i=n} x^i$$

Mais aussi par le schéma de récurrence suivant :

$$\begin{aligned} S(0) &= 1 \\ S(n) &= x \times S(n-1) + 1 \end{aligned}$$

On peut donc facilement construire le programme :

```
entier geo(N,X)
creation: S,K,SOMME
debut
  S:=1;
  K:=0;
  boucle N fois
    K:=mult(X,S);
    S:=S+1;
    SOMME:=add(SOMME,K);
   fin
 TMP:=retourner(SOMME);
fin
```

Suite de Fibonnacci

La suite de Fibonnacci est elle-aussi, une grande classique :

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

Ce schéma de récurrence n'est pas primitif récursif, mais on peut quand même écrire facilement la procédure en utilisant un autre schéma de récurrence.

$$Fibo(n, u, v) = Fibo(n-1, u+v, u-v).$$

Procédure :

```
entier fibo(N)
creation:U,V
debut
  U:=0;
  V:=1;
  boucle N fois
    V:=add(V,U);
    U:=diff(V,U);
  fin
 TMP:=retourner(U);
fin
```

On voit encore ici, la nécessité de l'instruction (4). En effet, sans elle, le calcul de la différence de V et de U ne serait plus linéaire et on aurait une complexité qui passerait de $O(F(n) \times n)$ en $O(F(n)^2 \times n)$.

4. Les fonctions de prédicat

4.1. Définitions.

DÉFINITION 6. *Un prédicat est une fonction dont les valeurs sont prises dans l'ensemble {vrai,faux}. L'ensemble des prédicats définis sur les naturels est donc*

$$\{N^k \to \{vrai, faux\} | 0 \geq k\}$$

Un prédicat P à k arguments est un sous-ensemble de N^k (les éléments de N^k pour lesquels P est vrai.

Par simplicité, nous allons choisir de représenter classiquement *vrai* par 1 et *faux* par 0 et considérer qu'un prédicat est défini par une fonction vers les entiers dont la valeur ne peut être que 0 ou 1.

DÉFINITION 7. *La fonction caractéristique d'un prédicat $P \subseteq N^k$ est la fonction $f : N^k \to 0, 1$ telle que*

$$f(\overline{n}) = \begin{cases} 0 & si \quad \overline{n} \notin P \\ 1 & si \quad \overline{n} \in P \end{cases}$$

La notion de prédicat primitif récursif est alors directement définie à partir de la notion de fonction caractéristique.

DÉFINITION 8. *Un prédicat est primitif récursif si sa fonction caractéristique est primitive récursive.*

4.2. Zéro. Le prédicat *zéro* qui n'est vrai que pour l'entier 0, est primitif récursif. En effet, sa fonction caractéristique *zerop* peut être définie par récursion primitive comme suit :

$$\begin{array}{lcl} zerop(0) & = & 1 \\ zerop(n+1) & = & 0 \end{array}$$

Procédure dans notre langage :

```
#Test l'égalité avec 0
entier zerop(X)
creation: Y
debut
 Y:=1;
 boucle X fois
   Y:=0;
   TMP:=sortir(X);
  fin
TMP:=retourner(Y);
fin
```

L'instruction "sortir" permet d'éviter de "boucler" pour rien. On gagne en complexité car on passe de $O(N)$ en un temps constant puisqu'on sort en un tour.

DÉFINITION 9. *L'instruction sortir(N) est une instruction qui permet de forcer la sortie d'une boucle (des appels récursifs) si et seulement si son paramètre N est supérieur à 0.*

Elle permet de ne pas boucler quand on a eu le résultat désiré. Elle permet donc de conserver la complexité de beaucoup d'algorithmes (et donc des programmes).

4.3. Signe. La fonction signe a la valeur 0 si son argument est 0 sinon 1. C'est le contraire de *zerop*.

$$\begin{array}{lcl} signe(m) & = & 0 \\ signe(m+1) & = & 1 \end{array}$$

Comme dans *zerop*, il est nécessaire, pour garder la complexité, d'utiliser la fonction "sortir".

```
#Test le signe d'un nombre
entier signe(X)
creation:Y
debut
 Y:=0;
 boucle X fois
   Y:=Y+1;
   TMP:=sortir(Y);
  fin
TMP:=retourner(Y);
fin
```

4.4. Plus petit. La fonction caractéristique du prédicat *plus petit* est définie par

$$petit(N, M) = signe(M - N)$$

Ce qui donne dans notre langage :

```
#Test si N est plus petit que M
entier pluspetit(M,N)
creation:X,Y
debut
  X:=diff(M,N);
  Y:=signe(X);
  TMP:=retourner(Y);
fin
```

4.5. Prédicats de la logique classique. Les prédicats obtenus à partir de prédicats primitifs récursifs par les opérations booléennes sont aussi primitifs récursifs. Soient $g_1(\overline{n})$ et $g_2(\overline{n})$ les fonctions caractéristiques de deux prédicats primitifs récursifs g_1 et g_2. Nous avons :

$$\begin{array}{lcl} et(g_1(\overline{n}), g_2(\overline{n})) & = & g_1(\overline{n}) \times g_1(\overline{n}) \\ ou(g_1(\overline{n}), g_2(\overline{n})) & = & signe(g_1(\overline{n}) + g_1(\overline{n})) \\ non(g_1(\overline{n})) & = & diff(1, g_1(\overline{n})) \end{array}$$

Les procédures, avec celles déjà écrites, sont élémentaires à écrire. **Procédure du "et" logique**

```
entier et(A,B)
creation:Z
debut
 Z:=mult(A,B);
 Z:=signe(Z);
 TMP:=retourner(Z);
fin
```

Procédure du "ou" logique

```
entier ou(A,B)
creation:Z
debut
 Z:=add(A,B);
 Z:=signe(Z);
 TMP:=retourner(Z);
fin
```

Procédure du "non" logique

```
entier non(A)
creation:Z
debut
 Z:=1;
 Z:=diff(Z,A);
 TMP:=retourner(Z);
fin
```

Avec ces prédicats, il est donc possible d'écrire tous les autres prédicats de la logique classique.

4.6. Egale. Le prédicat d'égalité (n=m) est lui aussi primitif récursif puisque $n = m$ si $\neg((n < m) \vee (m < n))$ ce qui se traduit au niveau des fonctions caractéristiques par

$$egal(n, m) \quad = \quad 1 - (signe(m - n) + signe(n - m))$$

Ce qui donne donc dans notre langage

```
entier egale(M,N)
creation:X,Y1,Y2
debut
  X:=diff(M,N);
  Y1:=signe(X);
  X:=diff(N,M);
```

```
  Y2:=signe(X);
  X:=add(Y1,Y2);
  Y2:=1;
  Y1:=diff(Y2,X);
  TMP:=retourner(Y1);
fin
```

4.7. IF THEN ELSE. Il est possible de simuler les "instructions conditionnelles" avec des boucles.

DÉFINITION 10. *Soit une fonction définie par :*

$$f(\overline{n}) = \begin{cases} g_1(\overline{n}) & si \quad p_1(\overline{n}) \\ \vdots & \\ g_l(\overline{n}) & si \quad p_l(\overline{n}) \end{cases}$$

où les fonctions $g_1, \cdots, g_l$ ainsi que les prédicats $p_1, \cdots, p_l$ sont primitifs récursifs, alors la fonction $f(\overline{n})$ est aussi primitive récursive. En effet elle est donnée par

$$f(\overline{n}) = g_1(\overline{n}) \times p_1(\overline{n}) + \cdots + g_l(\overline{n}) \times p_l(\overline{n})$$

Notons que pour que cette représentation soit correcte, il faut que les prédicats p_i soient mutuellement exclusifs.

DÉFINITION 11. *"IF A then P else Q" est primitif récursif.*

Justification : c'est un sous ensemble de la définition ci-dessus.
Algorithme :

```
debut
  SINON:=1;
  SI:=fonction_caractéristique(A)
  boucle SI fois
    P;
    SINON:=0;
   fin
  boucle SINON fois
     Q;
   fin
fin
```

5. D'autres algorithmes classiques

5.1. L'inférieur de deux nombres. Le problème du calcul de l'inférieur vient de la définition des fonctions primitives récursives : elles doivent nécessairement choisir un des arguments, ce qui a pour conséquence immédiate qu'aucun algorithme primitif récursif ne peut calculer la fonction Inf en un temps qui n'est fonction que du minimum et donc dans le bon "temps" (voir la Thése de L. Colson : *représentation intentionnelle d'algorithmes dans les systèmes fonctionnels.*

Pour trouver, l'inférieur de deux variables X et Y, il suffit de faire la différence de X et Y (Y-X) puis regarder si cette valeur est différente de 0. Si oui alors, le min est X sinon Y. Procédure :

```
entier inf(X,Y)
creation:RES
debut
  RES:=Y;
  Y:=diff(X,Y);
  boucle Y fois
    RES:=X;
    TMP:=sortir(Y);
   fin
```

```
 TMP:=retourner(RES);
fin
```

On utilise, l'instruction "sortir" pour gagner en complexité. En effet, cette instruction permet d'éviter de boucler encore pour rien (boucle Y fois...). La complexité est donc de O(X). Si, on n'avait pas l'instruction "sortir", on aurait une complexité en O(X+Y).

Dans un article récent, R. David a trouvé un algorithme n'utilisant que les instructions de base (1) (2) (3)) qui permet de contruire l'inf en un temps O(inf). L'algorithme n'est pas présenté ici, car cela dépasse notre sujet.

Une amélioration est toute fois possible en utilisant l'instruction de décrémentation et les "IF THE ELSE". En effet pour trouver l'inf de X et Y on peut décrémenter les deux à la fois en bouclant sur X. Ainsi, si dans la boucle Y devient Nul, c'est notre minimum. Sinon c'est X le minimum. On utilise donc massivement l'instruction permettant de forcer la sortie de boucle. Dans notre langage :

```
entier inf2(X,Y)
creation:MX,MY,SINON
debut
 MX:=X;
 MY:=Y;
 boucle X fois
  X:=X-1;
  Y:=Y-1;
  SINON:=1;
  boucle Y fois
   SINON:=0;
   TMP:=sortir(Y);
   fin
  boucle SINON fois
    TMP:=retourner(MY);
   fin
 fin
 TMP:=retourner(MX);
fin
```

La complexité est donc en $O(inf)$, puisqu'on sort si on trouve $Y \geq X$. Mais on "triche" par rapport à l'algorithme de R. David car on utilise des instructions qui sont normalement d'une complexité autre que constante.

5.2. Division euclidienne. Le problème de la division (euclidienne) de a par b est de déterminer deux entiers q et r tels que : $a = q \times b + r$ et $0 \geq r < b$.

Division et reste

Algorithme de la division de a par b (0 *geqa* et b>0) :

```
Def division(A,B) IS
debut
Q:=0;
R:=A;
 Si A<=B alors on a fini
  sinon
 boucle A fois
  R:=R-B;
  Q:=Q+1;
  si B<R alors sortir
 fin
fin {Q est le quotient et R le reste.}
```

Dans notre langage, on utilise alors deux variables globales, QUOTIENT et RESTE pour avoir ce couple (et donc pour ne pas faire deux algos différents pour chacun d'entre eux). La complexité de cet algorithme est normalement en $O(B)$ mais le calcul de la différence et du plus petit implique une complexité en $O(B^2)$. On remarque qu'il est nécessaire d'utiliser la fonction sortir car on ne connaît pas à l'avance le nombre d'itérations qu'il faudra effectuer.

Générateur de hasard à 1 pas

Les générateurs à un pas, permettent de simuler un hasard. Ils sont très rapides mais leurs pérodicités peuvent être facilement calculées (Brent,Floyd).

L'implémentation est facile, il suffit d'augmenter une variable globale GENERATEUR d'un PAS et de prendre le reste par une division euclidienne.

```
entier rand(PAS,MODULUS)
debut
 GENERATEUR:=add(GENERATEUR,PAS);
 TMP:=division(GENERATEUR,MODULUS);
 GENERATEUR:=RESTE;
 TMP:=retourner(GENERATEUR);
fin
```

5.3. PPCM. Le PPCM (plus petit commun multiple), peut lui aussi, être calculé avec le langage Loop. En effet, le maximum pour un PPCM de a et b est a*b. Pour garder, la complexité, il faut "sortir" quand on a trouvé le PPCM. Algo :

```
DEF ppcm(A,B) IS
MA:=A;
MB:=B;
boucle (A*B) fois
 si ma=mb alors sortir
  sinon
   si ma<mb alors ma:=ma+a
    sinon mb:=mb+b
fin
retourner(ma)
```

6. Multiplication et exponentiation dichotomique

Problème de complexité

Ces algos sont connus pour avoir une complexité logarithmique.

Comment alors avoir une complexité qui prenne ses valeurs dans les entiers ? Pour résoudre ce problème, nous allons boucler sur une borne supérieure et utiliser l'instruction "sortir" (La borne supérieure pourra être $|log_2(N)| + 1$, mais il est plus simple d'utiliser N).

6.1. La multiplication dichotomique. Elle est construite sur la formule de récurrence suivante :

$$\begin{array}{lcl} x \times y & = & x \times y + 0 \\ x \times 0 + p & = & p \\ x \times y + p & = & \left\{ \begin{array}{lll} 2x \times (y/2) + p & si & \text{y est pair} \\ 2x \times ((y-1)/2) + (p+x) & si & \text{y est impair} \end{array} \right. \end{array}$$

cette récurrence semble donc primitive récursive.
Mais il nous faut créer deux nouveaux prédicats :

- pair
- impair

6.2. Les prédicats pair et impair. Il existe deux manières de déterminer si un nombre est pair ou impair.

- Faire une division par 2 et regarder le reste (simple et utilisé en machine)
- Enlever deux un certain nombre de fois et quand on arrive à zéro (pair) ou à un (impair) on arrête

Algo :

```
R := N;
loop N do
 if R=1 then impair := 1; exit; endif
  else
   R := R - 1;
   R := R - 1;
   if R=0 then pair := 1; exit; endif
end loop;
```

Mais la première méthode,en *Loop*, par une division par deux sera grosso-modo identique. Du fait que toutes les procédures ont déjà été écrites, il est aisé de construire ces deux prédicats.
Algo de la multiplication $X \times Y$:

```
 A:=X;
 B:=Y;
 P:=0;
 loop B do
  if B=0 then exit; endif
  if B impair then P:=P+A; endif
  A:=2*A;
  B:=B div 2;
endloop
```

6.3. L'exponentiation dichotomique. L'exponentiation dichotomique fonctionne sur le même principe :

$$\begin{array}{lcl} x^y & = & x^y + 0 \\ x^0 \times p & = & p \\ x^y \times p & = & \begin{cases} (x^2)^{(y/2)} \times p & si \quad \text{y est pair} \\ (x^2)^{((y-1)/2)} \times p \times x & si \quad \text{y est impair} \end{cases} \end{array}$$

On peut donc de même écrire facilement l'algorithme :

```
A:=X;
B:=Y;
P:=1;
loop B do
 if B=0 then exit;endif
 if B impair then p:=p*a; endif#en utilisant la multiplication dichotomique
 A:=A*A;
 B:=B div 2;
endloop
```

CHAPITRE 8

Les tableaux

Introduction

Depuis, le début de cette partie, les éléments n'étaient que des entiers ≥ 0. Bien que nous n'expliciterons ni une définition mathématique des tableaux, ni leur intérêt, nous pouvons facilement les utiliser (si leur contenu est indicé de 0 à N) dans le langage *Loop* (et par conséquent dans le langage défini dans la première partie).

Comme précédemment, les algorithmes de base et donc les plus faciles sur les tableaux, sont facilement traductibles en *Loop* et donc auront leurs équivalences dans les fonctions primitives récursives.

1. Outils et séquentialité

Pour pouvoir utiliser les tableaux, deux outils de bases sont nécessaires
- la lecture d'un tableau
- l'affichage du tableau

Les algorithmes sont triviaux. Voici le code dans notre langage.

Lecture d'un tableau

```
#lecture  d'un tableau
entier liretab(Tab,TAILLE)
creation:A,LIRE
debut
 A:=1;
 boucle TAILLE fois
   LIRE:=lire();
   Tab[A]:=LIRE;
   A:=A+1;
 fin
fin
```

Affichage d'un tableau

```
#Afficher un tableau
entier affichetab(Tab,TAILLE)
creation:A,B
debut
 A:=1;
 boucle TAILLE fois
   B:=Tab[A];
   TMP:=ecrire(B);
   A:=A+1;
 fin
fin
```

Remarque : les tableaux sont indicés généralement de 1 à N, mais parfois il est bon de les indicer de 0 à N-1. Dans le langage décrit dans la première partie, les tableaux sont indicés de 0 à N, ce qui permet d'appliquer les deux méthodes.

1.1. Egalité de tableaux. L'égalité de tableaux est simple : on parcourt simultanément les deux tableaux et si on trouve une différence alors on sort. Code :

```
entier egalitedetablo(Tab1,Tab2,TAILLE)
creation: OU,TEST,A1,A2,SORT
debut
 OU:=1;
 TEST:=1;
 boucle TAILLE fois
  A1:=Tab1[OU];
  A2:=Tab2[OU];
  TEST:=egale(A1,A2);
  SORT:=non(TEST);
  TMP:=sortir(SORT);
  OU:=OU+1;
 fin
TMP:=retourner(TEST);
fin
```

1.2. Rechercher un élément. Si on considére que les tableaux permettent de garder en mémoire des données, il est bon de savoir si une donnée est présente ou non. Ceci est la recherche d'un élément dans un tableau. La recherche est ici séquentielle, nous verrons plus loin comment optimiser cette méthode. Code :

```
entier recherche(Tab,X,N)
creation:A,TEST,INC
debut
 INC:=1;
 boucle X fois
   A:=Tab[INC];
   TEST:=egale(A,N);
   boucle TEST fois
     TMP:=retourner(INC);
   fin
  INC:=INC+1;
 fin
fin
```

Dans le même registre, il est possible d'écrire :
- L'indice du minimum d'un tableau
- Valeur de cet élément
- Idem pour le maximal
- Insertion d'un élément
- Suppression d'un élément

Je ne le présente pas ici, car le code serait trop souvent redondant et n'apporterait rien de vraiment intéressant. Par contre, ces algorithmes ont été codés dans notre langage et ont été fournis dans un module.

Il est aussi aisé d'écrire l'échange de deux valeurs dans un tableau.

2. Les tris

Certain tris peuvent eux aussi être écrits en *Loop* (peut être par tous). Nous allons vous montrer les plus connus.

DÉFINITION 12. *tri : on considère une suite finie $X = (x_1, \cdots, x_n)$ d'éléments distincts appartenant à un ensemble muni d'un ordre total (ici les entiers positifs).*

La suite ordonnée $Y = (y_1, \cdots, y_n)$ correspondant à la suite X est une permutation de cette dernière vérifiant :

(1) $\forall i\ n \geq i \geq 1, \exists j, n \geq j \geq 1$ *tel que* $x_i = y_i$

(2) $\forall i\ n \geq i \geq 1$, *on a* $y_i \geq y_{i+1}$

2.1. Tri interne par selection du maximum. Cette méthode consiste à agrandir une partie ordonnée en selectionnant le maximal dans la partie non ordonnée. Code :

```
entier triselectmax(Tab,N)
creation:PAS,ECH,I,P
debut
 I:=N;
 boucle N fois
    P:=indicemaxtab(Tab,I);  #Trouver l'indice du maximal
    TMP:=echanger(Tab,I,P);  #Faire l'echange
    I:=I-1;
 fin
fin
```

2.2. Tri par Bulles. La méthode consiste à passer séquentiellement le tableau pour faire monter les éléments les plus "petits" jusqu'à ne plus faire de changement. Algo :

```
def tribulle(t:tableau,n:taille du dit tableau)
is
loop N do
  P:=0;
  SORT:=vrai;
  boucle N-1 do
     P:=P+1;
     if t[P]>t[p+1] then échanger(t[p],t[p+1]); SORT:=faux; endif;
   endloop
  if SORT then exit; endif
endloop
end
```

3. Recherche de motifs

La recherche de motifs peut être aussi implémentée dans en *Loop*. Les motifs sont bien entendu une suite non vide d'entiers comme le texte.

3.1. Méthode Naive. La méthode naive consiste à parcourir tout le texte, partie par partie en comparant chaque caractère avec ceux du motifs comme un tampon. On arrête la comparaison entre le motif et la partie de texte à la première différence. Si on ne trouve pas de différence, on a trouvé le motif et on l'affiche. Dans les deux cas, on décale le tampon (le motif) pour recommmencer. Il faut donc une procédure de comparaison entre le motif et la partie de texte (comparaison de tableau). Puis créer une procédure de première occurrence. Le programme sera une suite de recherches de première occurence. Source :

```
entier premiereoccurrence(P,Texte,LT,Motif,LM)
creation:I,LR,TROUVE,COMB
debut
 LR:=1; LR:=add(LR,LT); LR:=diff(LR,LM);
 I:=P; TROUVE:=comparer(I,Texte,Motif,LM);
 boucle TROUVE fois
   TMP:=retourner(I);
  fin
 COMB:=diff(LR,P);
```

```
 boucle COMB fois
   I:=I+1;
   TROUVE:=comparer(I,Texte,Motif,LM);
   TMP:=sortir(TROUVE);
  fin
 TROUVE:=non(TROUVE);
 boucle TROUVE fois
   I:=LR;
   I:=I+1;
  fin
 TMP:=retourner(I);
fin
```

Autre Algorithmes

Il est aussi possible d'implémenter la méthode "PSN" (pas si naive) dans notre langage et bien d'autre algorithmes de recherche de motifs. Mais ceci dépasse notre sujet.

4. Recherche dichotomique

La recherche dichotomique permet dans un tableau ordonné de trouvé un élément en complexité logarithmique.

4.1. Méthode, rechercher l'entier x. Si l'intervalle de recherche est réduit à un singleton, on compare x et l'élément correspondant.
Sinon, comparer x avec l'élément médian. Puis selon le résultat, recommencer à la recherche en considérant soit l'intervalle incluant la médiane, soit l'intervalle supérieur à la médiane.
Comme pour la multiplication et l'exponentiation dichotomique, il va falloir boucler sur une borne supérieure.

4.2. Algorithme.

```
IndiceDicho(tab t[n], x:Element)
begin
if (n=0) then i:=0
 else
   begin
     j:=1;
     k:=n;
     loop n do
       if (j<k) then exit;
       m:=(j+k) div 2;
       si (x<=t[m]) then k:=m;
        else
          j:=m+1;
      endloop
    if x=t[j] then i:=j
     else i:=0;
   end
return i;
end
```

On voit encore ici, l'intérêt d'une instruction de sortie de boucle. En effet sans celle-ci, on perdrait la complexité logarithmique pour une complexité linéaire.

Avec cette recherche dichotomique, il est possible de faire un tri par insertion (non exposé ici).

5. Les polynômes

Les polynômes peuvent être représentés par des tableaux. Le langage *Loop* est donc bien adapté au traitement des algorithmes sur les polynômes.

La recherche du degré est la recherche (en partant par la fin) dans le tableau du premier élément non nul (voir précédemment).

5.1. Somme. La somme de deux polynômes p et q, s'écrit par la formule suivante :

$$s_i = p_i + q_i \text{pour tout } 0 \geq i \geq N.$$

La récurrence est donc immédiate. Algo :

```
begin
 I:=0;
 S[0]:=p[0]+q[0];
 loop N do
  I:=I+1;
  s[I]:=p[I]+q[I];
 endloop
end
```

5.2. Produit. Le produit de deux polynômes P et Q s'écrit mathématiquement :

$$\forall n \in \mathcal{N} R_n = \sum_{k+l=n} a_k \times b_l$$

Il s'agit donc d'une double itération sur les indices des deux polynômes (les deux tableaux). L'algo est donc tout simplement :

```
#P de degres N, Q de degres M donc R d degres N+M
begin
 I:=0;
 loop N do
   J:=0;
   loop M do
     R[I+J]:=R[I+J]+P[I]*Q[J]
     J:=J+1;
    endloop
   I:=I+1;
  endloop
end
```

5.3. Valeur en X, méthode de Horner. L'évaluation d'un polynôme en X peut être effectuée à l'aide de la méthode par rangs décroissants (basée sur le schéma de Horner). La récurrence est la suivante :

$$k > 0, q = \sum_{i=k}^{n} a_i \times x^{i-k}$$

Il est aisé de voir que ce schéma de récurrence est primitif récursif (suite finie d'additions et de multiplications).
Il est donc aisé de le transcrire dans notre langage :

```
entier horner(Poly,N,X)
creation:H,K,A
debut
 H:=Poly[N];
 K:=N;
 boucle N fois
  K:=K-1; A:=Poly[K];
  H:=mult(H,X); H:=add(H,A);
 fin
 TMP:=retourner(H);
fin
```

THÉORÈME 3. *(Borodine, 1973), L'algorithme de Horner est "optimal".*

CHAPITRE 9

Les matrices et les graphes

Introduction
Les matrices peuvent être facilement manipulées dans notre langage. Les algorithmes associés auront donc leurs équivalences en fonctions primitives récursives. Les graphes sont représentables par une matrice d'adjacence. Certains algorithmes sur les graphes pourront donc être écris en *Loop*. Tous les algorithmes sont bien sûr des algorithmes de base.

1. Les matrices

1.1. Somme. La somme de deux matrices A et B, s'effectue par la somme de tous les éléments des deux matrices respectives. Procédure dans notre langage :

```
entier addmatrice(Mat1,N,M,Mat2,Mat3)
creation: TMP1,TMP2,I,J,OU
debut
 I:=0;
 boucle N fois
    I:=I+1;
    J:=0;
    boucle M fois
      J:=J+1;
      OU:=placer(I,J,M);
      TMP1:=Mat1[OU];
      TMP2:=Mat2[OU];
      TMP1:=add(TMP1,TMP2);
      Mat3[OU]:=TMP1;
     fin
 fin
fin
```

La complexité de l'algo est en $O(N * M)$. Mais pour accéder aux éléments, nous avons une complexité en $O(I * M + J)$. Ce qui donne une complexité en $O(N * M^2)$.

1.2. Produit. Le produit de deux matrices $A \times B = C$ s'écrit mathématiquement par la formule suivante :

$$c_{ij} = \sum_{k=1}^{n} b_{ik} \times a_{kj}$$

Chaque entité de la matrice s'obtient à partir d'une suite finie de multiplications et d'addtions. Il est donc facile d'obtenir un algorithme en *Loop*.

```
#On suppose A de dimension(M,N) et B de dimension(N,P)
begin
 I:=0;
 loop N do
  I:=I+1;
  J:=0;
  loop P do
    J:=J+1;
    C[I,J]:=0;
    K:=0;
    loop M do
```

```
      K:=K+1;
      C[I,J]:=C[I,J]+A[I,K]*B[K,J];
     endloop
   endloop
 endloop
end
```

La complexité est normalement en $O(N*M*P)$. Mais les additions et les multiplications n'ont pas dans notre cas une complexité constante. Il faut donc ajouter à cette complexité le calcul des C[I,J]. La complexité reste pour autant polynômiale.

1.3. Transposition. La transposée d'une matrice s'obtient avec la formule suisante : Soit $A = (a_{ij})\ (i,j) \in I \times J$. La transposée de A est la matrice $B = (b_{kl})\ (k,l) \in J \times I$ tel que $\forall (k,l) \in J \times I\ b_{kl} = a_{lk}$.

Il existe deux méthodes pour calculer la transposition d'une matrice. La première copie les valeurs dans une matrice. Ceci peut être effectué par un simple parcours des éléments de la matrice. La seconde calcule la transposée directement dans la matrice. Algo :

```
begin
 K:=1;
 loop N do
   I:=0;
   loop K do
     I:=I+1;
     echanger(A[K+1,I],A[I,K+1]);
   endloop
 enloop
end
```

Comme précédement, l'algorithme est polynômial.

1.4. Inversion. L'inverse d'une matrice A est la matrice B, notée A^{-1}, tel que $A \times A^{-1} = I$ ou I est la matrice Identité.

Cette inverse peut être obtenue par pivotages successifs sur les éléments de la diagonale de la matrice et en effectuant les mêmes opérations sur une matrice Identité. Après ces pivotages, l'ex-matrice identité est l'inverse recherchée.

Pivotage dans une matrice

On suppose que la matrice est de dimension (M,N) et que l'on veut pivoter à l'élément indicé (R,S). Algo :

```
begin
 J:=1;
 loop N do
  M[J,R]:=M[J,R] / M[S,R];
  J:=J+1;
 endloop
 I:=0;
 #Les éléments avant R
 loop R do
   J:=0;
   loop N do
     M[J,I]:=M[J,I]-M[S,I]*M[J,R];
    endloop
   I:=I+1;
 endloop
 I:=I+1;
 #Les éléments après R
 loop (M-R) do
```

```
   J:=0;
   loop N do
     M[J,I]:=M[J,I]-M[S,I]*M[J,R];
    endloop
   I:=I+1;
 endloop
end
```

1.5. Algorithme de l'inversion. Maintenant il suffit de faire les pivotages successifs sur la concaténation de la matrice A avec I (nommé M).

```
begin
 I:=0;
 loop M do
  pivoter(M,I,I);
 endloop
end
```

Remarque : La division utilisée est bien entendu la division euclidienne. Il n'est donc pas possible d'utiliser réellement cette matrice, car les erreurs d'approximation sont bien trop importantes.

1.6. Représentation et utilisation. Les matrices peuvent êre représentées par un tableaux à une seule dimension. En effet si on a une matrice M de dimension M*N alors pour atteindre M[I,J] il suffit de faire Tab[I*N+J] pour accéder à cet élément. On peut ainsi avec une procédure appropriée, accéder facilement à tous les éléments d'une matrice.

L'utilisation de matrice est donc très simple. Par exemple, pour afficher une matrice, il suffit d'afficher dans l'ordre tous les éléments du tableaux. Pour la lecture, il suffit de boucler sur la "largeur" et la "longueur".

2. Les graphes orientés

Introduction

Je ne ferai pas de description sur les graphes ni sur la preuve et le fonctionnement des algortihmes. Le but ici est de montrer que les algorithmes de bases sur les graphes peuvent être transposés dans notre langage et donc ont leurs équivalences dans les fonction primitive récursive.

Les graphes sont bien entendu représentés par leurs matrice d'adjacence

Nous présenterons deux principaux algorithmes pour la recherche du plus court chemin.

2.1. Algorithme de Floyd. L'algorithme de Floyd permet de construire la matrice suivante :

$$C^k_{i,j} = \begin{cases} \text{coût minimum d'un chemin ne passant par aucun sommet intermédiaire} \\ \text{infini s'il n'existe pas} \end{cases}$$

Ceci peut être fait par le schéma de récursion suivant :

$$C^0_{i,j} = \begin{cases} \text{0si i=j} \\ \text{A(i,j) si i<>j} \end{cases}$$

$$C^m_{i,j} = min \begin{cases} C^{k-1}_{i,j} \\ C^{k-1}_{i,k} + C^{k-1}_{k,j} \end{cases}$$

On peut voir que ce schéma est primitif récursif. Il est donc possible d'écrire un algorithme en *Loop* (car la fonction min est primitive récursive).

```
begin
 I:=0;
 loop n do
  I:=I+1; J:=0;
  loop n do
    J:=J+1; C[I,J]:=A[I,J];
   endloop
```

```
  C[I,I]:=0;
 endloop

 K:=0;
 loop n do
   K:=K+1; I:=0;
   loop n do
     I:=I+1; J:=0;
     loop n do
       J:=J+1; C[I,J]:=min(C[I,J],C[I,K]+C[K,J]);
      endloop
    endloop
  endloop
end
```

On pourrait avec le même genre d'algorithme écrire celui de Warshall.

2.2. Dijkstra. L'algorithme de Dijkstra permet de calculer les plus courts chemins à partir d'une origine. On obtient donc le tableau suivant :

$$D_t(k) = \begin{cases} \text{coût minimum d'un chemin de s à x ne passant par un sommet intermédiaire hors de T} \\ \text{infini s'il n'existe pas} \end{cases}$$

Algo grossier :

```
Mat: notre graphe
Dg: tablo[sommet] de valeur initialiser avec Mat
Te: tablo[sommet] de boolean tous a faux

Dijkstra(s:sommet)
begin
 initialiser(Dg,Cg,Te);
 Te[s]:=vrai; Dg[s]:=0;
 loop N-1 do
   t:=pluspetit(Dg);    /* plus petit élément d'un tableau*/
   Te[t]:=vrai;
   tt:=s0;
   loop N do
   if (not(Te[tt])) then
     if Dg[t]+G[t,tt]<D[tt] then begin
                               D[tt]:=D[t]+G[t,tt];
      Cg[tt]:=t;
     end
   tt:=tt+1;
  end
 end
end
```

3. Fonction primitive récursive $=>$ *Loop*

Des recherches sont encore en cours pour avoir des transformations automatiques de programmes *Loop* en fonctions primitives récursives et vice-versa. L'intérêt évident de cette partie sera qu'après tranformation de ces algorithmes en fonctions primitives récursives, les preuves pourront être effectués plus facilement (démonstrateur automatique).

4. Algorithmes difficilles

Le QuickSort est le premier exemple d'algorithme qui est difficilement traductible en *Loop* à causes des appels récursifs. Le simplex est lui aussi un algorithme qui parraît difficilement traductible. En effet, il n'est pas aisé de savoir combien de fois il va boucler.

Conclusion

Nous venons d'écrire un large ensemble d'algorithmes[1] et de programmes qui ont leurs équivalences dans les fonctions primitives récursives.

Dans un projet pédagogique, il apparaît alors que l'itération non bornée soit un outils bien trop puissant pour les algorithmes de base. De plus, la terminaison étant syntaxiquement prouvée, le programmeur est soulagé d'attendre vainement la fin de l'exécution de son programme.

On peut aussi dégager de ce travail que beaucoup d'algorithmes sont basés sur un schéma de récurrence primitif récursif. Le langage *Loop* (et les langages de programmation basés sur les boucles) apparaît alors comme un outils suffisant, tant du point de vue de "l'expressivité" que du point de vue de la "complexité".

Mais,la question qui va être traitée dans la partie suivante est :

Peut-on faire mieux ?

[1] Pour plus d'informations sur ces algortihmes, veuillez lire le livre [2] de la bibliographie.

Troisième partie

Les Hyperloops

Introduction

Dans la dernière partie, les programmes implémentant les algorithmes avaient une terminaison syntaxiquement prouvée. Mais, on ne pouvait pas construire "plus" que les algorithmes basés sur les fonctions primitives récursives.

En effet, certaines fonctions, qui finissent, ne sont pas primitives récursives. D'autre fonctions, bien qu'étant primitives récursives,comme la fonction de Fibonnacci, ont des algorithmes qui les calcules qui ne pourront pas être écrit en *Loop*.

Pour certaines d'entre elles, il est cependant possible de construire un autre algorithme en *Loop* , sur une autre schéma de récursion, permettant de les calculer. Mais quand est t'il des fonctions non primitives récursive ?

Il apparaît alors, la nécessité d'introduire une nouvelle forme de boucle pour remédier à ce problème.

Celle-ci est nommée dans la littérature : l'*hyperloop* ou *powerloop*. Elle a fait son apparition dans le langage Madcap dans les années 60. Mais ses avantages n'ont jamais été réellement décrits et cette instruction ne fit que de rares apparitions (article de Mandl, Funkel) pour décrire des ensembles d'algorithmes.

Nous allons nous efforcer de montrer que cette nouvelle forme de boucle permet la construction de programmes qui, syntaxiquement, se finissent toujours et permettent de capturer des algorithmes que le langage *Loop* ne pouvait pas.

CHAPITRE 10

Introduction de l'hyperloop

1. Premier exemple : l'exponentiation

Dans la première partie, nous avons pu écrire l'addition, la multiplication et l'exponentiation et les coder. Chaque algorithme nécessitait l'introduction d'une boucle supplémentaire. L'exponentiation avait une boucle sur une variable "non constante", c'est-à-dire qui changeait de valeur au cours de l'exécution. Ne pourrait-on pas éviter cette boucle ? La définition mathématique de l'exponentiation est la suivante :

$$M^N = \overbrace{M \times M \times M \cdots \times M}^{N \text{ fois}}$$

Or on sait que la multiplication de $M \times M$ peut s'écrire en *Loop*

```
begin
 res:=0;
 loop M do
   loop M do
     res:=res+1;
   endloop
 endloop
 return res.
end
```

Il apparaît donc qu'on pourrait écrire l'exponentiation de la manière suivante

```
begin
  loop M do
    loop M do
```

N-2 fois {

```
      ...
      loop M do
       res :=res+1 ;
      endloop
      ...
```

}

```
    endloop
  endloop
 return res.
end
```

Cet algorithme n'est pas possible avec le langage *Loop* puisqu'on ne connaît pas à l'avance le nombre de boucles nécessaires. On remarque, que chaque boucle est imbriquée dans une autre boucle (exceptée la première). On peut donc introduire l'hyperloop de cette manière :

```
global:res,N
local:M
begin
 res:=0;
 hyperloop N do
   loop M do
     deeper;
   endloop
 endhyper
```

```
  do final
   res:=res+1;
  endfinal
return res;
end
```

L'instruction *deeper* a pour effet de "relancer" le code.

Quand on relance trop, l'hyperloop déclenche ce qu'on pourrait appeler le *final* :"res :=res+1", ceci pour éviter de relancer une infinité de fois le code.

2. Sémantique informelle

2.1. Réduction. Nous allons dans cette section, essayer de décrire une sémantique opérationnelle de l'hyperloop.

On nomme par $\overrightarrow{X}$, les variables d'environnement d'exécution des instructions (les variables locales de l'algorithme et les compteurs de boucle).
Soit le code suivant :

```
hyperloop N do
   P(X);
   deeper;
   Q(X);
endhyper
 do final
   K(X);
 endfinal;
```

se réduit suivant la valeur de N

Réduction dans le cas ou N=0
C'est le cas du "final" donc

```
K(X);
```

Réduction dans le cas ou N>0
Le code est relancé au "deeper".
donc nous avons :

```
P(X);
hyperloop (N-1,X) do
   P(X);
   deeper;
   Q(X);
 endhyper
   do final
     K(X);
   endfinal
Q(X);
```

C'est à dire plus simplement :

```
hyperloop N do
 P(X)
 hyperloop(N-1,X)
 Q(X)
endhyper
```

Ainsi à chaque "deeper", on sauve l'environnement qui sera ensuite restauré. On retrouve ainsi, en Q, les valeurs de $\overrightarrow{X}$ d'avant le "deeper". On ne tient pas compte des changements de valeur faits dans les "sous hyperloops" (sauf bien sûr des variables globales).

Remarque : dans les articles traitant des hyperloops (ou powerloop), voir [3] [4] [5] de la bibliographie, les auteurs exigeaient un "deeper" et un seul pour chaque hyperloop. Nous allons voir dans le chapitre suivant que cette exigence n'est pas nécessaire.

2.2. Exemple simple avec deux "deeper". soit le code :

```
hyperloop N do
   A;
   deeper;
   B;
   deeper;
   C;
endhyper
 do final
   K;
 endfinal;
```

Pour $N = 2$, la machine exécutera dans l'ordre :

```
A
| A
| | K
| B
| | K
| C
B
| A
| | K
| B
| | K
| C
C
```

3. Un autre exemple, la factorielle

Nous avons vu que la factorielle pouvait se calculer avec *Loop*. Rappelons la formule :

$$N! = \overbrace{1 \times 2 \times 3 \times 4 \cdots (N-1) \times N}^{N \text{ fois}}$$

On voit ici qu'il faut N boucles imbriquées pour faire les N multiplications.
Algo :

```
global:N,RES
local:X
begin
 hyperloop N do
  X:=X+1;
  loop X do
   deeper;
   endloop
 endhyper
 do final
  RES:=RES+1;
 endfinal

return(res);
end
```

Ce qui donne :

```
begin
  loop 1 do
    loop 2 do
N-2 fois { ...
           loop N do
            res :=res+1 ;
           endloop
           ...
    endloop
  endloop
 return res.
end
```

Ce qui correspond bien à la définition du début.

Remarque : le fait de sauvegarder l'environnement, permet d'avoir le résultat désiré car x n'est pas modifié au retour de l'hyperloop du dessous

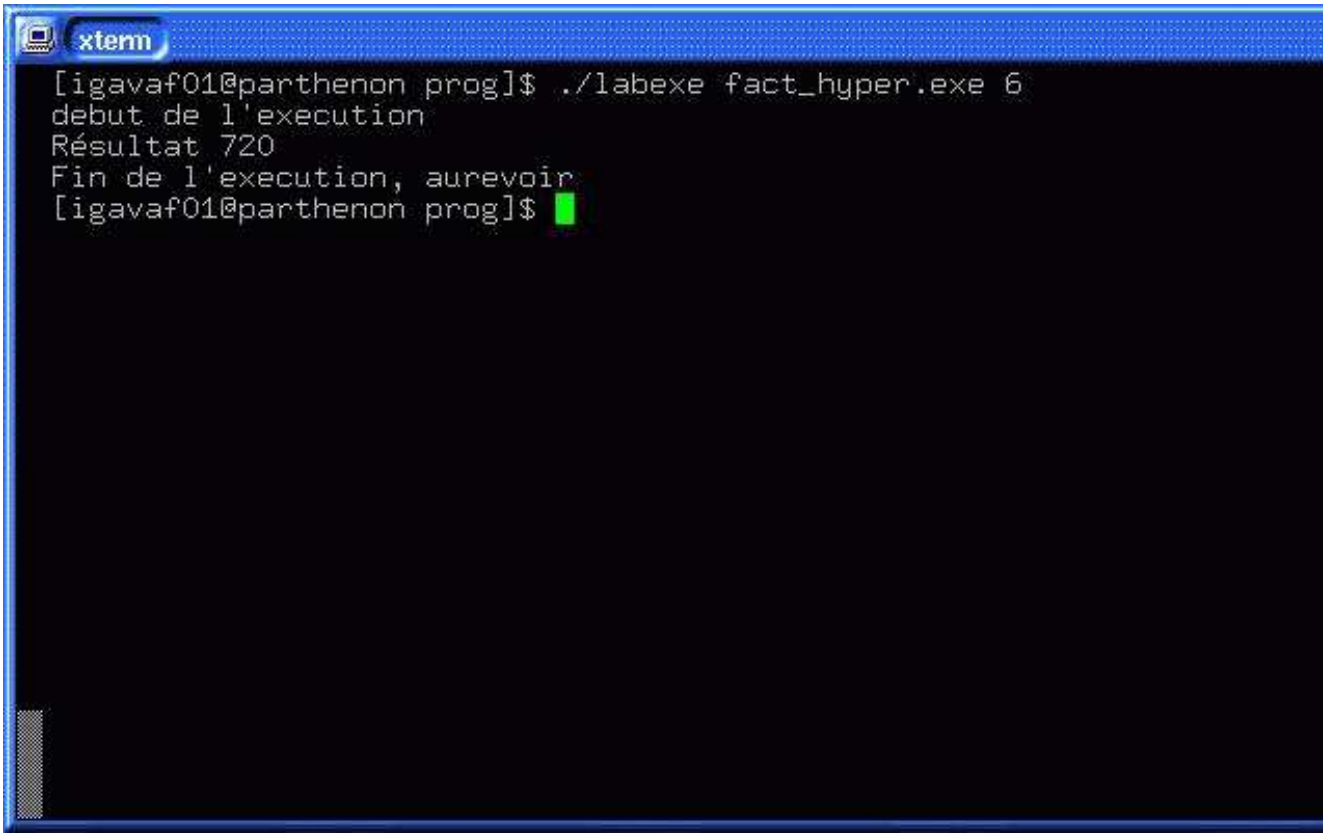

FIG. 1. Exemple de la factorielle en hyperloop en LAB

CHAPITRE 11

Implémentation machine

0.1. Explication Générale. L'implémentation en machine peut sembler difficile, mais la solution est en réalité très simple grâce à une pile et à des compteurs.

En effet, nous avons vu précédemment qu'il fallait sauvegarder l'environement à chaque "deeper" puis relancer le code. Sauvegarder implique d'empiler les variables locales et les compteurs de boucles. Restaurer sera l'exacte contraire : dépiler.
Le problème est le suivant : où faut il restaurer l'environnement ?

Une première solution, serait de restaurer l'environnement après chaque "deeper". Le code serait très vite volumineux car on serait obligé de faire autant de "restauration" que de "deeper"

La solution retenue est que l'environnement soit restauré à la fin du code de l'hyperloop (au endhyper) et à la fin du "final" (au endfinal). Ainsi, quand on revient au précédent "deeper", on a fini soit un "final" soit une "sous hyperloop" et dans les deux cas l'environnement a été restauré. On gagne ainsi en code car on ne restaure au plus que deux fois l'environnement.

Le second problème est comment détecter "le final" et la fin de l'hyperboucle. La solution qui a été retenue et qui semble la meilleure (en temps et en espace) est d'utiliser deux compteurs. Le premier est décrémenté à chaque "deeper" et incrémenter à chaque restaurations d'environnement. Ainsi, quand ce compteur est à zéro, on est sûr d'être dans le cas du final. Le second est une constante pour ne pas trop incrémenter ce premier compteur et arrêter l'exécution de l'hyperloop.

0.2. Schéma de l'implémentation. Pour une hyperloop sur "N", on a le schéma suivant :

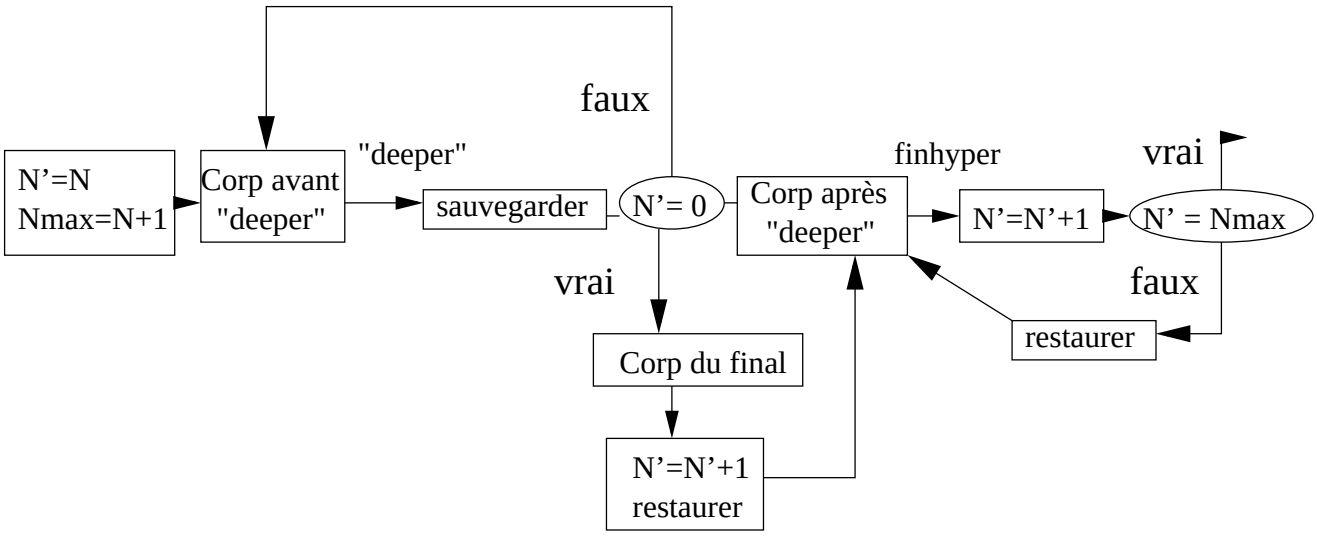

FIG. 1. Implémentation des hyperloops, cas d'un deeper

Remarque : si l'hyperboucle contient plusieurs "deepers", le schéma avec "l'avant deeper" et "l'après deeper" se repète autant de fois qu'il y a de "deepers"..

On voit donc qu'il est parfaitement possible d'utiliser plusieurs "deeper" sans que cela gène le bon fonctionnement des programmes.

Bien entendu, dans la sauvegarde (et par conséquent la restauration) de l'environnement, il faut, en langage machine, empiler le pointeur de code, pour ultérieurement, revenir au bon endroit (après le deeper).

0.3. Résultat dans notre langage machine. Exemple avec 2 "deepers".

```
hyperloop N do
  A;
  deeper;
  B;
  deeper
  C;
 endhyper
 do final
  K;
 endfinal
Suite...
```

Donnerait en code intermédiaire :

```
mov Nmax N+1          | 3) C
mov N' N              | si N'=Nmax jump en 5)
1) A                  | pop pointeur
push environnement    | pop environnement
push 2)               | jump en pointeur
dec N'                | 4) K
si N'=0 jump en 4)    | inc N'
jump en 1)            | pop pointeur
2) B                  | pop environnement
push environnement    | jump en pointeur
push 3)               | 5) Suite...
dec N'                |
si N'=0 jump en 4)    |
jump en 1)            |
```

Remarque : nous insistons sur le fait que l'on ne sauvegarde que les objets locaux. Ainsi, les objets globaux ne sont pas affectés par le deeper. Ceci permet une "communication" entre les différentes hyperloops.

Remarque : Dans le cas d'hyperloop imbriquées, a quelle hyperloop fera référence le "deeper" ? Deux solutions sont envisageables :

(1) soit chaque hyperloop et chaque deeper sont numérotés

(2) soit le deeper fait référence à la dernière hyperloop. (solution retenue)

0.4. Francisation dans notre langage. Pour garder une cohérence avec le langage décrit dans la première partie, nous pouvons écrire la syntaxe de l'hyperloop de la manière suivante :

```
hyperboucle N fois
  A;
  TMP:=enfoncer();
  B;
  TMP:=enfoncer();
  C;
 end
```

```
final
  K;
end
```

Cette syntaxe est claire et précise. La fonction "enfoncer" correspond au "deeper". Elle retourne toujours la valeur 0.

CHAPITRE 12

Backtracking, Hanoi et Fibonnacie

1. Backtracking

Je ne présenterais pas ici, le principe du backtracking mais plutôt une manière de l'écrire avec les hyperloops.

1.1. Principe général. En effet, les hyperloops ont été initialement conçues pour traiter des problèmes de backtracking. L'algorithme générique est le suivant :

```
hyperloop N do
 if Somecondition(Level) then deeper
  else write("failed at level");
 endhyper
 do final
  write("succes");
 endfinal
```

les hyperloops que nous avons décries précédement se prêtent donc parfaitement à ce schéma et donc aux algorithmes de backtracking.

1.2. Le problème des huit reines. le problème des huit reines est un grand classique du backtracking. Le problème est de poser huit reine sur un échiquier (de 8*8 cases) sans que celles-ci ne puissent s'attraper (voir les règles des Jeux d'échecs). L'algorithme grossier que nous pourrions écrire, en le généralisant à N reines, est donc le suivant :

```
local:Column
begin
 hyperloop N do
   Columm:=1;
   loop N do
      if oksofar(Queen[Columm]) then deeper
    endloop
   endhyper
     do final
       write(Queen[1..N])
     endfinal
end
```

Qui peut se décomposer, pour N=3, ainsi :

```
 loop N do
  if oksofar(Queen[1]) then
   loop N do
    if oksofar(Queen[2]) then
       loop N do
        if oksofar(Queen[3]) then write(Queen[1..3]);
       endloop
    endloop
 endloop
```

Bien sûr une implémentation dans notre langage nécessite plus de structures de données, notamment pour les tests de poses des reines. On peut aussi remarquer la nécessité des objets globaux

(les tableaux mémorisant l'emplacement des reines etc...). Voici le résultat dans notre implémentation.

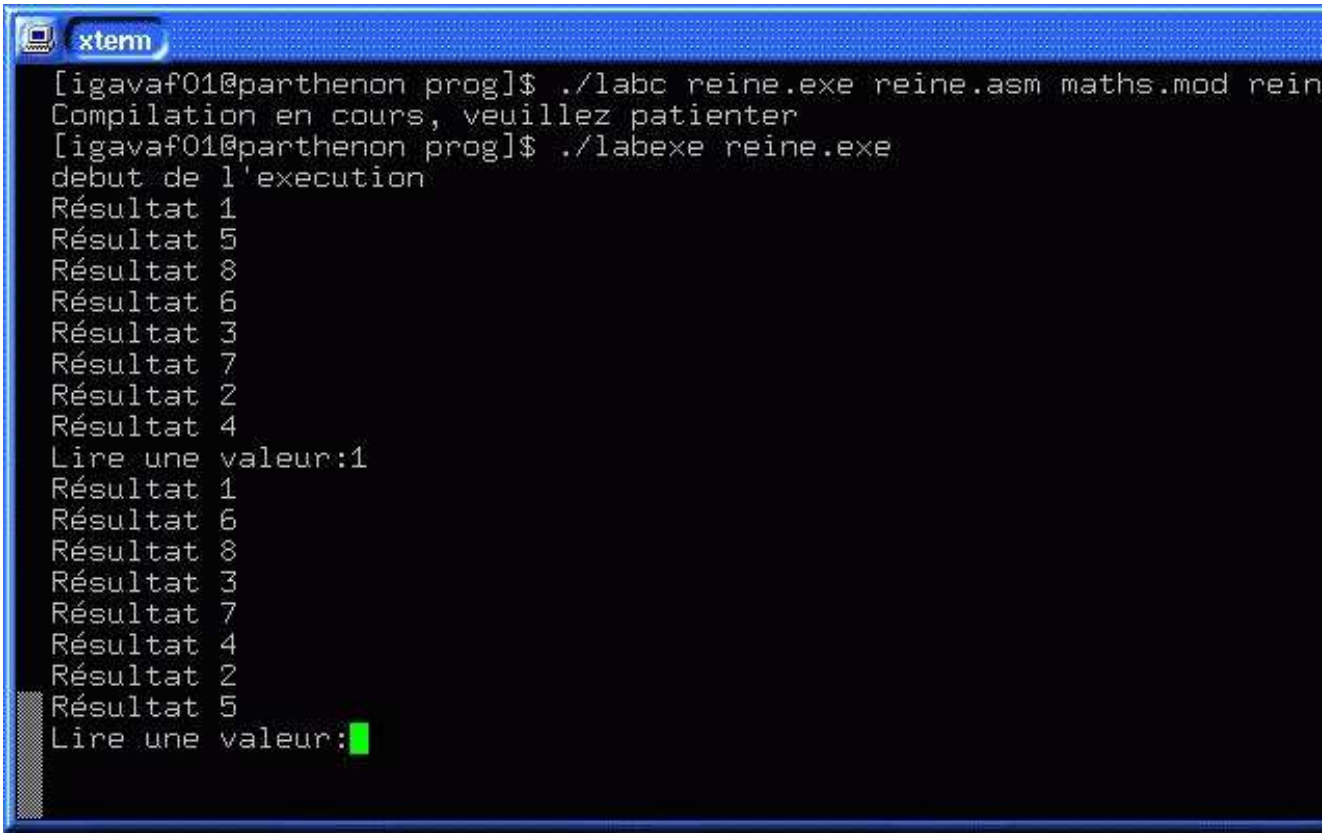

FIG. 1. Exemple des huit reines

2. Les tours de Hanoi

Les "tours de Hanoi", ne sont pas un problème de backtracking mais permettent de voir la récursivité dans les hyperloops. Le problème des tours de Hanoi est de déplacer N disques d'un piler vers un autre en utilisant un pilier auxiliaire. On ne peut déplacer qu'un seul disque à la fois sans le poser sur un disque plus petit que lui.
Algo :

```
local:dest,inter,source
begin
inter:=2;
source:=1;
dest:=3;
N:=N-1;  #histoire d'arriver plus vite à un final qui ne soit pas vide.
hyperloop N do
 INTER:=6-(source+dest);
 exchange(inter,dest);
 deeper;
 exchange(inter,dest) #pour remettre dans l'ordre
 write(source,dest);
 exchange(source,inter);
 deeper;
endhyper
 do final
```

```
 write(source,dest);
endfinal
```

la complexité est connue pour être en $O(2^N)$. Exemple de résultats dans notre langage :

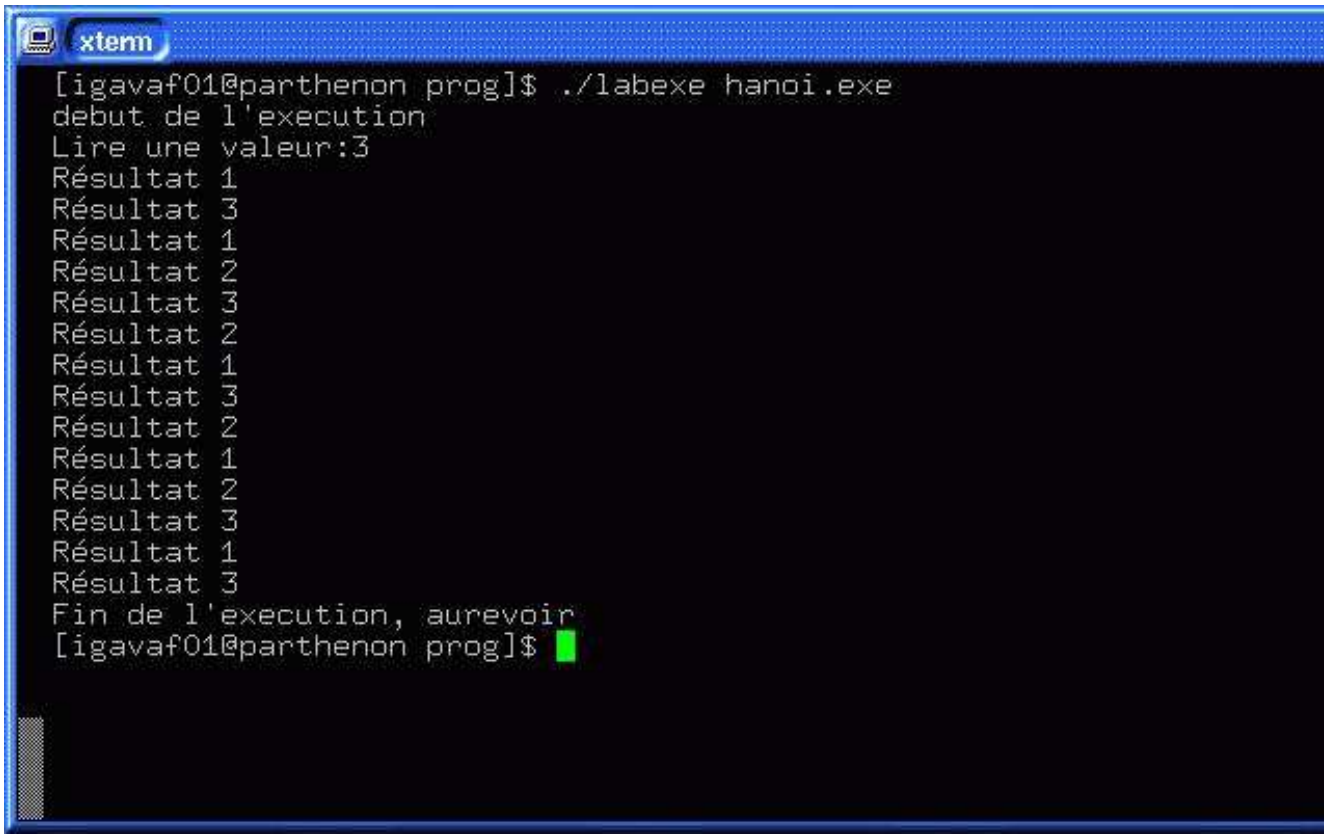

FIG. 2. Les tours de Hanoi, (tour de départ, tour d'arrivée)...

3. La fonction de Fibonnacci

Celle-ci peut être calculé de la manière suivante :

$$\begin{aligned} Fibo(0) &= 1 \\ Fibo(1) &= 1 \\ Fibo(n) &= Fibo(n-1) + Fibo(n-2) \end{aligned}$$

Il faut noter que ceci n'est pas la définition mais un algorithme. Le schéma de récurrence associée à cet algorithme n'est pas primitif récursif et donc l'algorithme ne pourra jamais être écrit en *Loop*.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F(n)	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

Schéma :

L'algorithme pourrait être implémenté en *Hyperloop* de la manière suivante :

```
global:res
local:N,F
hyperloop N do
 N:=N-1;
 if (N=0) then RES:=1; #N=0 ou N=1
  else
   begin
     RES:=0;
```

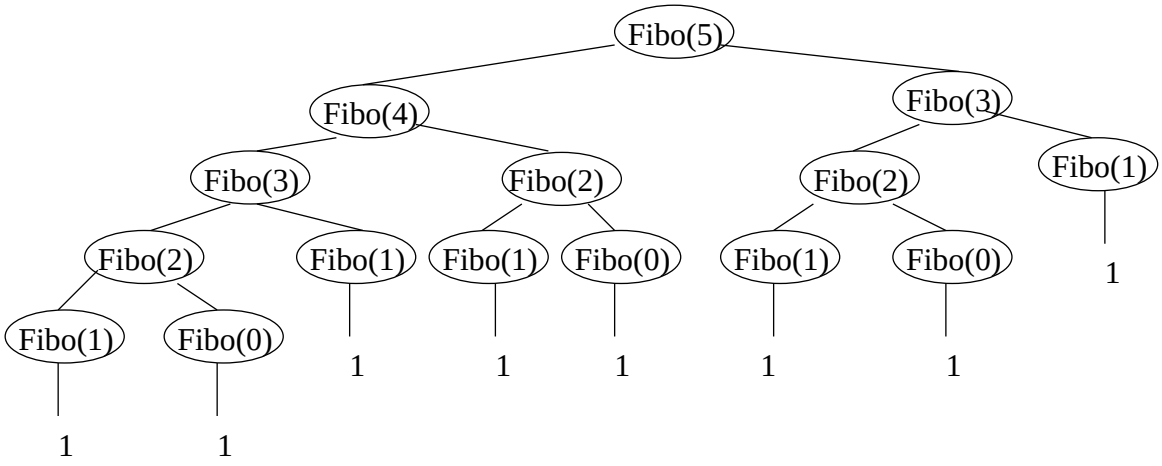

FIG. 3. Exemple de la fonction de Fibonnacie

```
     deeper; #calcul de F(N-1);
     F:=RES;
     N:=N-1;
     deeper;
     res:=res+f
   end;
 endhyper
do final
endfinal
return res
end
```

Il faudrait maintenant prouver formellement que l'algorithme en hyperloop calcule bien la fonction de Fibonnacie. Le problème est qu'il n'existe pas de système de preuve sur les hyperloops (logique à la Hoare).

Conclusion

Ces 3 algorithmes classiques ne sont pas directement transductibles en *Loop* (en généralisant les huit reines aux N-reines) sans utiliser une panoplie d'outils annexes. Ainsi, si ces algorithmes et la construction des hyperloops sont corrects, peut-on alors parler du langage *Hyperloop*?

CHAPITRE 13

Une première approche théoriques

Introduction
Nous allons, dans ce chapitre, énumérer un certain nombre d'idées sur le langage des hyperloops.

1. Premiers résultats

1.1. Equivalence de langage.

LEMME 2. *Loop* $\in$ *Hyperloop*

Justification :

Elle se fait en transformant une boucle en une hyperboucle de la manière suivante :

```
loop N do
  P;
endloop
```

Peut facilement se transformer de la manière suivante :

```
hyperloop N do
 P;
 deeper;
endhyper
do final
 endfinal
```

ou avec un final

```
hyperloop (N-1) do
 P;
 deeper;
endhyper
do final
  P
endfinal
```

1.2. Terminaison.

THÉORÈME 4. *Tout programme* $\in$ *Loop termine.*

Justification : les fonctions primitives terminent toujours. Donc par équivalence les programmes *Loop* aussi.

LEMME 3. *Tout programme* $\in$ *Hyperboucle termine.*

Justification :
La "relance" de code due aux "deepers" est bornée par le "final". On sait qu'un "deeper" est équivalent à $hyperloop(N-1, \overrightarrow{X})$ ou au final. La suite $N := N - 1$ est une suite bornée par 0 (le final) donc qui termine. On peut donc en dire autant des hyperloops.

1.3. Questions.

CONJECTURE 1. *Hyperloop* $\notin$ *Loop*

Les hyperloops pourraient elles calculer plus que les fonctions primitives récursives ?

Si dans les exemples précèdents (Hanoi, Huit reines, Fibonnacci) les algorithmes sont vérifiés alors les hyperloops permettraient déjà de construire des algorithmes qui ne sont pas possible en *Loop*.

La question que nous pouvons nous poser est la suivante : Quelle est la classe de fonctions que calculent les *hyperloops* ?

Nous allons dans le chapitre suivant essayer d'apporter un début de réponse à cette question avec une fonction très connue : la fonction d'Ackermann.

CHAPITRE 14

La fonction d'Ackermann

Introduction
L'intérêt de cette fonction est double : c'est une fonction non primitive récursive qui termine. Cela signifie qu'on ne peut pas l'écrire dans notre langage *Loop*. Cette fonction peut être vue (modulo codage) comme un interpréteur de programmes primitifs récursifs ; pour voir cela, il suffit de construire un interpréteur de combinateurs primitifs récursifs pour voir apparaître dans le programme le schéma de la fonction d'Ackermann.

Si nous pouvions écrire une algorithme et un programme permettant de calculer cette fonction, on aurait la preuve de la conjoncture.

1. Construction fonctionnelle et "récursive"

Nous avons pour la multiplication la définition suivante :

$$\begin{array}{lcl} N \times 0 & = & 0 \\ N \times (M+1) & = & N + (N \times M) \end{array}$$

Similairement, la fonction puissance (n^m) se définit à partir de la multiplication :

$$\begin{array}{lcl} N^0 & = & 1 \\ N^{M+1} & = & N \times N^M \end{array}$$

Par analogie avec la définition de la fonction puissance par récursion primitive à partir de la fonction produit, il est possible de définir une nouvelle fonction en appliquant la récursion primitive à la fonction puissance : la fonction :$n \uparrow\uparrow m$ définie par :

$$\begin{array}{lcl} N \uparrow\uparrow 0 & = & 1 \\ N \uparrow\uparrow (M+1) & = & N^{\uparrow\uparrow M} \end{array}$$

est primitive récursive. Elle est équivalente à :

$$N \uparrow\uparrow M = \left. N^{N^{N^{N^{N^{\cdots^{N}}}}}} \right\} \text{M fois}$$

la fonction $n \uparrow\uparrow\uparrow m$ est définie par :

$$\begin{array}{lcl} N \uparrow\uparrow 0 & = & 1 \\ N \uparrow\uparrow\uparrow (M+1) & = & N \uparrow\uparrow N \uparrow\uparrow M \end{array}$$

est aussi primitive récursive. Il en va de même de toutes les fonctions n $(\uparrow)^k$ (k ≥ 2) définies par

$$\begin{array}{lcl} N(\uparrow)^k 0 & = & 1 \\ N(\uparrow)^k (M+1) & = & N(\uparrow)^{k-1}(N(\uparrow)^k M) \end{array}$$

Dans la définition précédente, on peut considérer k comme un argument de la fonction plutôt que comme un paramètre définissant une famille de fonctions. Nous avons donc :

$$f(k+1, n, m+1) = f(k, n, f(k+1, n, m))$$

En ne considérant plus le paramètre n qui ne joue aucun rôle dans la récursion on obtient (en remplacant k par m et m par n) la définition de la fonction d'Ackermann

$$\begin{array}{lcl} Ack(0, n) & = & n+1 \\ Ack(m+1, 0) & = & Ack(m, 1) \\ Ack(m+1, n+1) & = & Ack(m, Ack(m+1, n)) \end{array}$$

La fonction d'Ackermann, est la première fonction non primitive récursive, qui est Turing calculable et qui termine. Elle grossit plus vite que n'importe quelle fonction primitive récursive (voir table).

Ack(m,n)	**n=0**	**n=1**	**n=2**	**n=3**	**n=4**	**n=5**	**n=6**	**n=7**	**n=8**	**n=9**
m=0	1	2	3	4	5	6	7	8	9	10
m=1	2	3	4	5	6	7	8	9	10	11
m=2	3	5	7	9	11	13	15	17	19	21
m=3	5	13	29	61	125	253	509	1021	2045	4093
m=4	13	65533	!	!	!	!	!	!	!	!
m=5	65533	!	!	!	!	!	!	!	!	!

Les "!" correspondent à des valeurs trop importantes.

Même si ces valeurs sont très vite importantes, nous avons la propriété suivant :

PROPRIÉTÉ 1. *$\forall M \forall N$, Ackermann(M,N) termine.*

Preuve :
On montre la propriété suivante :

$$P(m) = \text{ pout tout } n, Ack(m,n) \text{ termine}$$

Preuve par récurrence sur m :

$m = 0$**:** trivial

on suppose $P(m)$ **vraie (HR1):** il faut donc montrer la propriété $P(m+1)$. On le montre par récurrence sur n. On montre la propriété suivante :

$$Q(n) = Ack(m+1,n) \text{ termine}$$

$n = 0$**:** $Ack(m+1,0) = Ack(m,1)$ termine par hypothèse de récurrence

on suppose pour tout n $Q(n)$ **vraie(HR2):** il faut alors montrer $Q(n+1)$ termine avec comme hypothèse de récurrence (HR2).

$$\begin{aligned} Q(n+1) &= Ack(m+1,n+1) \\ &= Ack(m, Ack(m+1,n)) \text{ par définition} \\ &= Ack(m,N) \text{ par HR2 } Ack(m+1,n) \text{ termine} \\ &= M \text{ par HR1 } Ack(m,\alpha) \text{ termine} \end{aligned}$$

On peut donc en conclure la propriété.

2. Description du problème

2.1. Premières propriétés.

PROPRIÉTÉ 2. *La fonction d'Ackermann ne peut être calculée par un programme Loop.*

Preuve : la fonction d'Akermann n'est pas primitive récursive donc par équivalence entre les fonctions primitives récursives et les programmes *Loop*, le théorème est vérifié.
Donc on a : $\nexists$(Il n'existe pas) $P \in Loop$ $\forall M \forall N$ tel que $P(M,N) = Acker(M,N)$.Un programme ayant un nombre de boucles FIXE ne peut calculer Ackermann.

Par contre nous avons :

(1) $\forall M \forall N \exists P \in Loop$ tel que $P(M,N) = Acker(M,N)$ En effet il suffit de faire

```
X:=0;
loop Acker(M,N) do
    X:=X+1;
 endloop
```

On peut le faire puisqu'on connaît la valeur en M,N (fixés).

(2) On sait que A(0,N)=N+1 et A(1,N)=N+2 et A(2,N)=2N+3 et $A(3,N) = 2^{N+3} - 3$ etc... donc par définition de la fonction d'Ackermann $\forall M$ $\exists P \in Loop$ $\forall N$ tel que$P(M,N) = Acker(M,N)$ (M fixé).

2.2. Rappel de théorèmes du calcul formel.

```
....
for i1:=a1 to b1 do
  for i2:=a2 to b2 do
    ....
      for ik:=ak to bk do
        P;
      end
    ....
  end
end
```

THÉORÈME 5. *Si les $\forall j$ les a_j,b_j sont fixes alors le nombre d'itérations est*

$$\prod_{j=1}^{k}(b_j - a_j + 1)$$

THÉORÈME 6. *Si les a_j sont des constantes et les b_j des polynômes en les $a_l, l < j$ alors le nombre d'itérations de P est un polynôme en les bornes supérieures.*

D'après ce théorème, on peut penser que les programmes *Loop* ne peuvent calculer que des polynômes. Par contre quand les b_j ne sont pas des polynômes alors il est possible de faire plus. Exemple : factorielle,exponentiation, etc...

L'idée
Nous avons vu, que la fonction d'Ackermann est un interpréteur de programmes primitifs récursifs. Elles doit donc faire varier son nombre de boucles pour "interpréter" un programme.

L'idée est que si un programme *Loop* en faisant varier ses indices de boucles peut calculer plus que des polynômes alors un programme *Hyperloop*, (dont le nombre de boucles varie) en faisant varier les indices des boucles peut-ils calculer la fonction d'Ackermann ? Si oui, alors la conjecture(différence entre *Loop* et *Hyperloop*) sera vérifiée.

2.3. Le problème. Le problème est donc :
Un programme ayant un nombre fini mais VARIABLE de boucles (nombre indéterminé) peut-il calculer Ackermann ? $<=> \exists ? P \in Hyperloop\ \forall M \forall N$ tel que $P(M,N) = Acker(M,N)$.

3. Construction impérative

Nous allons dans cet section, décrire comment construire un algorithme permettant de calculer la fonction d'Ackermann.

3.1. Premiers résultats.

DÉFINITION 13. *$\forall k$, L_k est la classe des fonctions calculables par $Loop_k$ (k boucles imbriquées).*

LEMME 4. (1) *$\bigcup_{n=0}^{+\infty} L_n$ est la classe des fonctions primitives récursives.*
(2) *$\forall n$, L_n est fermé par la composition.*
(3) *$\forall n \geq 2$, L_n est fermé par if-then-else et la minimisation bornée.*

Si l'on note $g^{(0)}(x) = x$ et $g^{(n+1)}(x) = g^{(n)}(g(x))$ alors on peut prouver le théorème suivant :

THÉORÈME 7. $L_n \subset L_{n+1}$

Ce théorème énonce qu'il y a des fonctions qui nécessitent un minimum d'itération bornée imbriquée. Et cela à chaque niveau. La fonction candidate à la preuve de ce théorème est :

$$\begin{array}{lcl} f_0(0) & = & 1 \\ f_0(1) & = & 2 \\ f_0(x) & = & x+2 \text{ pour } x > 1 \\ f_{n+1}(x) & = & f_n^{(x)}(1) \end{array}$$

LEMME 5. $f_{n+1}(x+1) = f_n(f_{n+1}(x))$

THÉORÈME 8. *Pour tout* $n \geq 1, f_n \in L_n$.

preuve : Par induction sur n.

$$\begin{array}{rcl} f_1(0) & = & 1 \\ f_1(x) & = & 2x \text{ pour } x > 0 \end{array}$$

Un programme qui calcule f_1
Le programme suivant calcule f_1 :

```
y := 1;
loop x do
   x := x + 1;
   y := x;
end loop;
```

par définition $f_{k+1}(x) = f_k^{(x)}(1)$ alors le programme suivant calcule $f_{k+1}(x)$:
Un programme pour $f_{k+1}(x)$

```
y := y + 1;
z := z + 1;
loop x do
   y := $f_{k}(z)$;
   z := y;
end loop;
```

THÉORÈME 9. $f_{n+1} \in L_{n+1} - L_n$

THÉORÈME 10. *Si on pose* $A(n,x) = f_n(x)$*, alors on a que* $A(n,x)$ *n'est pas primitive récursive.*

Il faut bien comprendre que $f_n(x)$ est une fonction qui ne dépend pas vraiment de n.
Dépliage du programme précédent

```
def f(k, x) is
begin
   y := y + 1;
   z := z + 1;
   loop x do
      y' := y' + 1;
      z' := z' + 1;
      loop z do
         y' := f(k-2,z');
         z' := y'
      end loop;
      y := y';
      z := y;
   end loop;
end
```

On voit bien ici les boucles qui s'imbriquent et les indices des boucles qui varient. On peut aussi constater l'affectation à chaque fin de boucle. Ces trois choses vont être à la base de l'algorithme en *Hyperloop*.

3.2. Solution(?) en *Hyperloop*. Voici une solution que je propose :

```
Global:RES
integer Ackermann(M,N)
create:ACK
begin
```

```
  RES:=N;
  if (M=0) then return(RES+1);

  hyperloop M do
     ACK:=RES+1;
     RES:=1;
     loop ACK do
        deeper;
     endloop
   endhyper
     final
       RES:=RES+1;
     endfinal

 return(RES);
end
```

La complexité semble être :

$$\sum_{i=0}^{i=M}\sum_{j=0}^{j=N} Acker(i,j).$$

Mais reste encore à vérifier.

3.3. Résultats pratiques. J'ai vérifié l'algorithme (en l'implémentant dans le Langage de la première partie) sur le tableau précédent.
Pour obtenir Ack(3,9), il a fallu à mon PC (K6-2 300 MHZ), environ 1 minute et 10 secondes.

Pour Ackermann(5,0)=65533, il faudrait environ 10 heures, ce qui est vraiment trop. Cette lenteur est du à la gestion de la pile de ma machine virtuelle. La pile étant un tableau dynamique, l'allocation (et la désallocation) de la mémoire ralentissent considéralement l'exécution.

J'ai donc utilisé pour vérifier Ack(4,1)=Ack(5,0)=Ack(3,13), le programme C suivant :

```
#include <stdio.h>
int RES;
void ack(int M)
{
 int I,A;
 if (M==0) RES++;
  else
   {
    A=RES+1;
    RES=1;
    for(I=1;I<=A;I++) ack(M-1);
   }
}
int main()
{
 int M,N;
 M=5;
 N=0;
 RES=N;
 ack(M);
 printf("Ack(%d,%d) = %d\n",M,N,RES);
}
```

Ce programme en C calcule Ackermann(5,0) en moins de 2 minutes alors qu'en utilisant la méthode classique (sur le schéma de récursion), il a fallu plus de 10 minutes. Ce programme semble

donc bien plus efficace.

3.4. Conjecture.

CONJECTURE 2. *L'algorithme calcule la fonction d'Ackermann*

Une première preuve serait de trouver le même schéma de récurrence dans l'article [7] de la bibliographie. Malheureusement, celui-ci n'a pas encore été trouvé.
Vaine tentative de justification :

Nous allons essayer de voir que de l'algorithme semble bien calculer la fonction d'Ackermann. Ceci n'est absolument pas une preuve mais c'est la seule chose qui a été trouvée pour l'instant. Elle figure dans ce texte pour représenter les recherches qui ont été faites.

Pour comprendre il faut bien voir que les M sont les niveaux de l'hyperboucle.
On cherche la propriété suivante :

$$P(M) = \text{pour tout} N, Algo(M,N) = Ack(M,N).$$

Récurrence sur M.

Cas Ack(0,N): Trivial, c'est le cas de base traité au début de l'algorithme.

Cas Ack(1,N)=n+2: En dépliant l'algorithme on obtient

```
RES:=N;
ACK:=RES+1    --donc Ack=N+1
RES:=1;
loop ACK do
 RES:=RES+1;
end
```

Ce qui équivaut à faire

```
RES:=1;
loop (N+1) do
  RES:=RES+1;
endloop
```

Ce qui donne bien N+2.

Cas Ack(2,N)=2*N+3: En dépliant l'algorithme on obtient

```
RES:=N;
ACK:=RES+1;
RES:=1;
loop ACK do
  ACK':=RES+1;
  RES:=1;
  loop ACK' do
     RES:=RES+1;
   endloop
endloop
```

ce qui équivaut à

```
RES:=1;
loop (N+1) do
  A:=RES+1;
  RES:=1;
  loop A do
    RES:=RES+1
   endloop
endloop
```

(au augmente de 2 en 2 N+1 fois à partir de 1)
Ce qui donne bien (2*N)+3.

on suppose $P(M)$ vraie (HR1): il faut donc montrer la propriété $P(M+1)$. On le montre par récurrence sur N. On montre la propriété suivante :

$$Q(N) = Algo(M+1, N) = Ack(M+1, N)$$

$N = 0$**:** $Ack(m+1, 0) = Ack(m, 1)$ donc l'algo calcule bien par hypothèse de récurrence.

On suppose pour tout N $Q(N)$ **vraie:** il faut alors montrer $Q(N+1)$ avec comme hypothèse de récurrence (HR2).

$$\begin{aligned} Q(N+1) &= Ack(M+1, N+1) \\ &= Ack(M, Ack(M+1, N)) \text{ par définition} \\ &= Ack(M, \alpha) \text{ par (HR2) } \alpha = Ack(M+1, N), Algo(M+1, N+1) = Ack(M, Ack(M+1, N)) \\ &= M' \text{ par HR1 } Algo(M+1, N+1) = Ack(M+1, N+1) \end{aligned}$$

Le calcul de α est la valeur RES (précédement calculer) après le dernier "deeper". Nous allons donc boucler sur cette valeur avec N'=N-1 puisqu'on trouve de nouveaux un "deeper".(voir schéma global) en remettant RES à 1. On demande donc à calculer Ack(M,α). Exemple : Ack(4,1) demande le calcul de Ack(4,0), celui-ci retourne 13 et on calcul donc Ack(3,13).

Dépliage

```
RES:=N;
ACK:=RES+1;
RES:=1;
loop ACK do
 ACK':=RES+1;
 RES:=1;
 loop ACK' do
   ACK'':=RES+1;
   RES:=1;
   loop ACK'' do
     ACK''':=RES+1;
     RES:=1;
     loop ACK''' do
       ...
         RES:=RES+1
       ...
      endloop
    endloop
  endloop
endloop
```

Il est facile de voir que le nombre de boucles varie et que leurs itérations grossit vite car on boucle sur une valeur non constante.

On peut donc "supposer" que notre algorithme calcule bien la fonction d'Ackermann. Reste à trouver un contre-exemple, trouver une preuve formelle ou prouver le contraire.

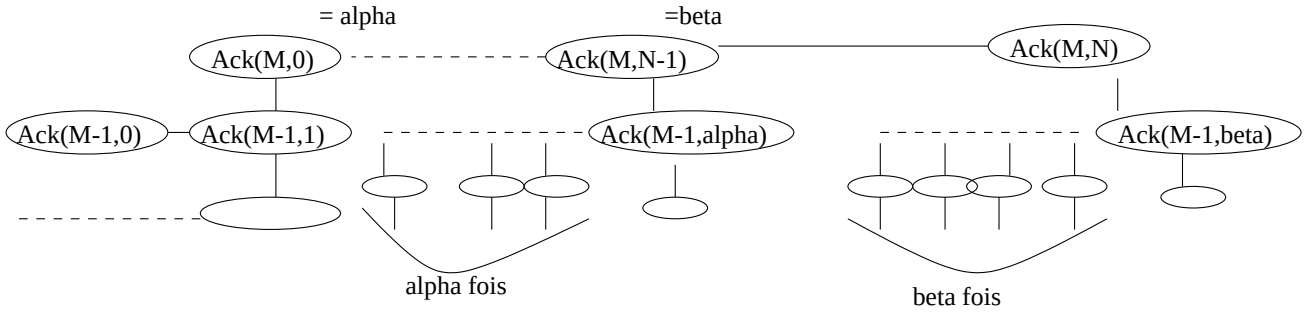

FIG. 1. Fonction Ackermann, cas général

Conclusion

Le compilateur ,malgrès de gros défaults (pas de vrais modules, aucune optimisation de code, pas de linker etc...) peut être utilisé par des étudiants et des programmeurs(surtout pour les hyperloops et sa syntaxe clair). La terminaison syntaxique des programmes, permet de soulager l'étudiant quand il doit faire des preuves. De plus, ce langage est très portable.

Avec cette séries d'algorithmes, il apparaît que le langage *Loop* est bien plus imporant qu'il ne le parait. En rajoutant, quelques instructions (décrémentations, sortie de boucles), il permet de conserver une bonne complexité.

Les hyperloop permettent la construction d'algorithmes aussi naturelement que les simples boucles. Pourrait-elles faire plus que les boucles ? Aucune preuve n'a été faite pour l'instant, mais cette preuve sera un prochain sujet de recherche.

Ce travail a été effectué sous la tutelle de Pierre Valarcher, Maître de Conférences à l'Université de Rouen (Laboratoire d'Informatique Fondamentale et Applications).

ANNEXE A

Exemple de la fonction d'Ackermann

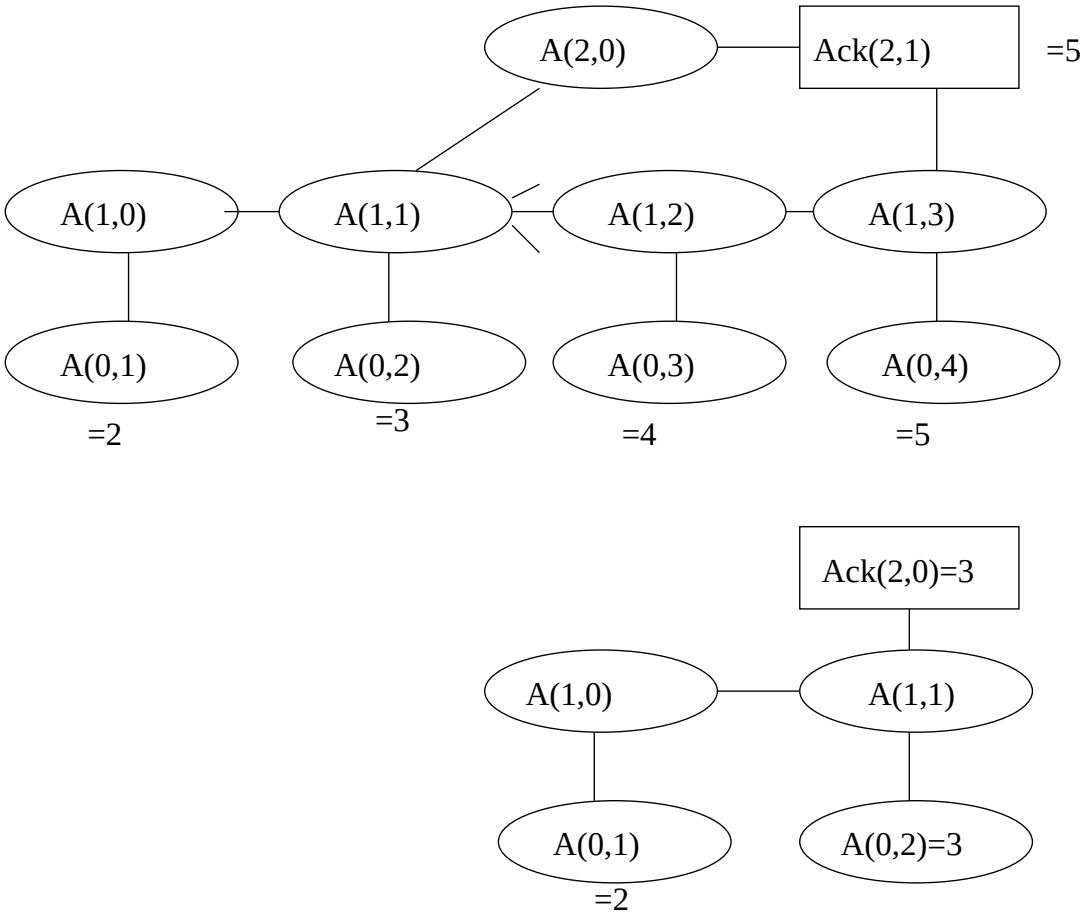

FIG. 1. Exemple Ackermann

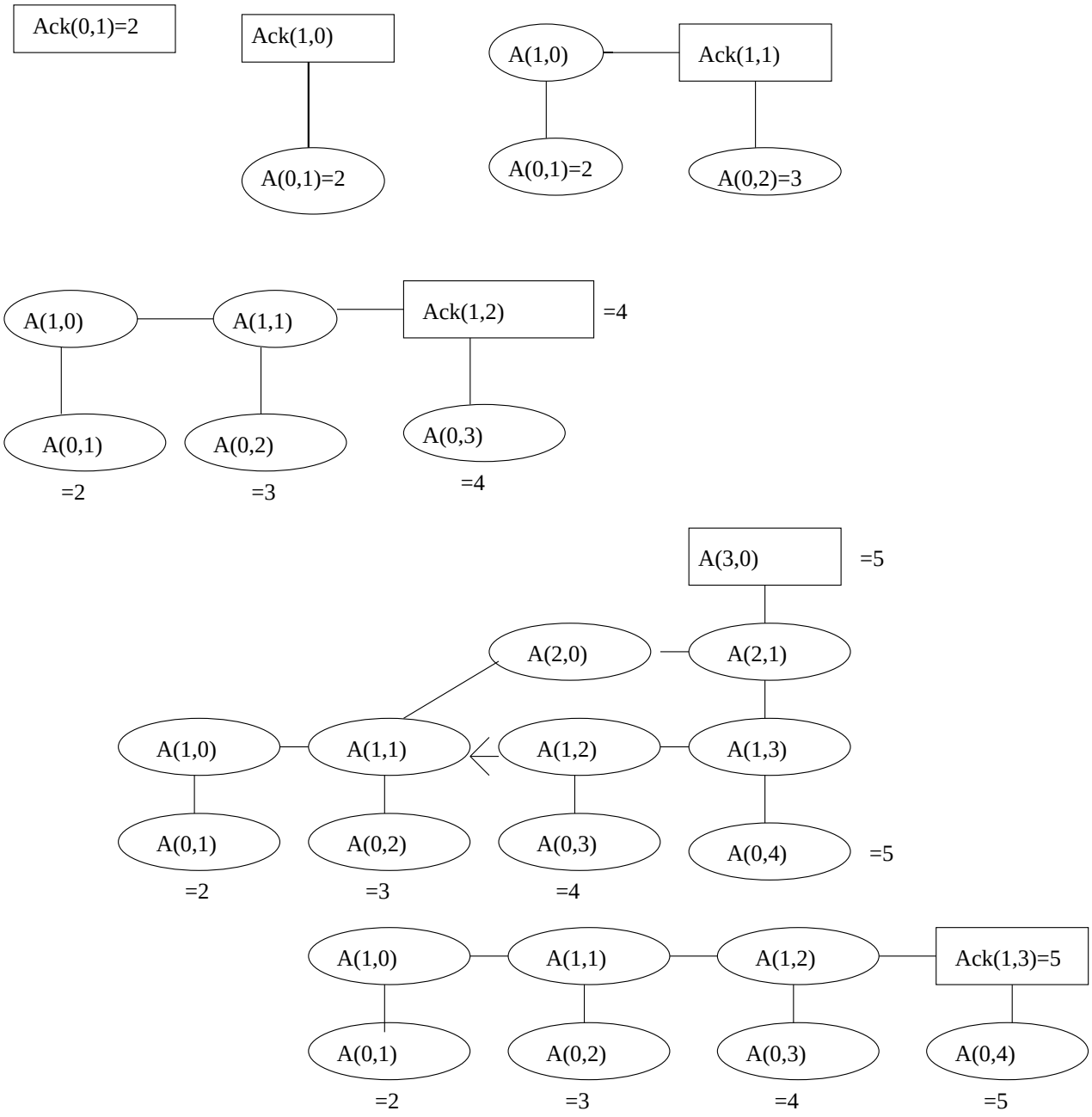

FIG. 2. Suite exemple Ackermann

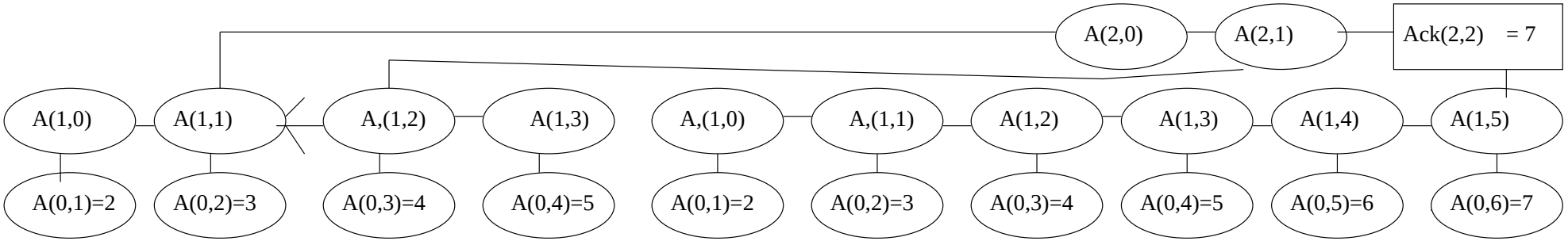

FIG. 3. Suite exemple Ackermann

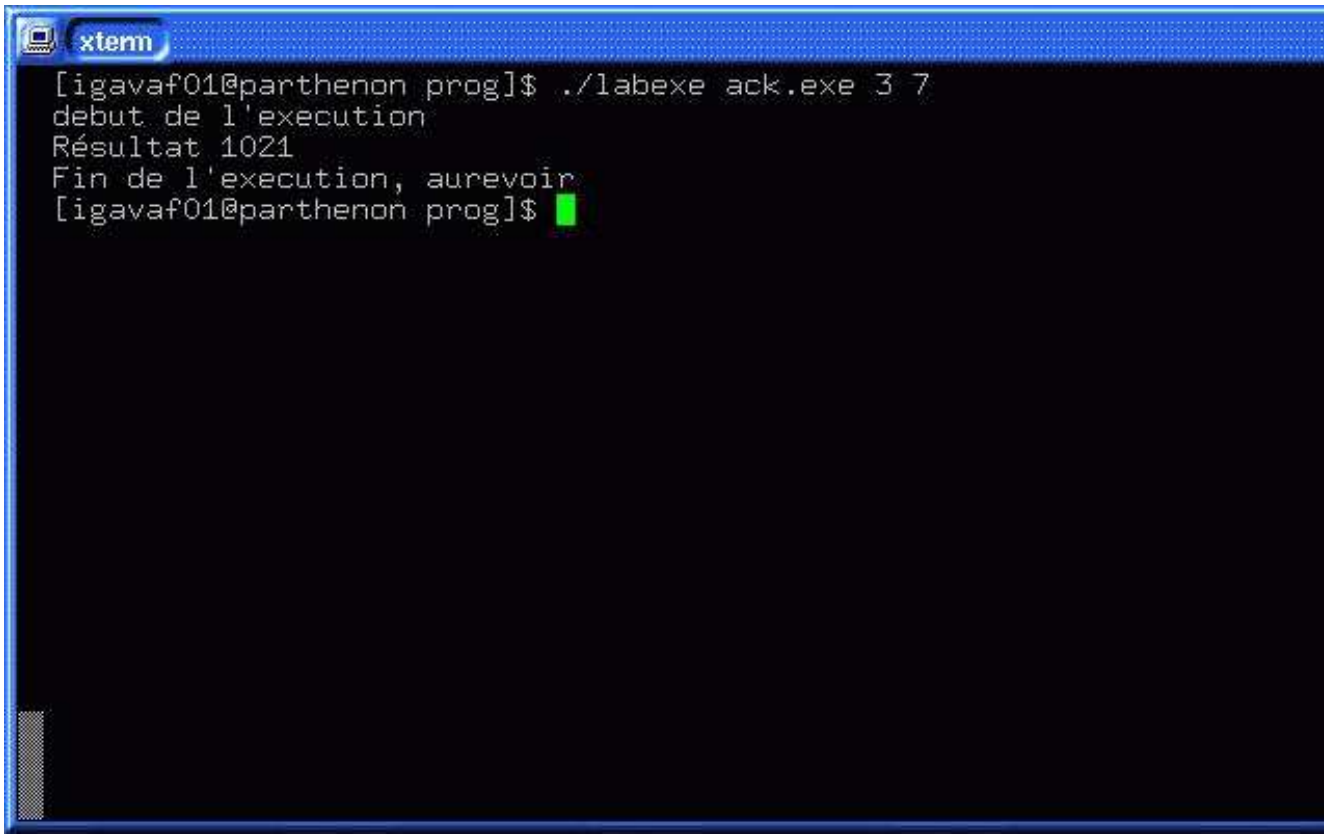

FIG. 4. Exemple de la fonction Ackermann dans l'environnement LAB

ANNEXE B

La Grammaire Complète du langage

Fichier -> Modules Programme

Global -> CREATION DEUX_POINTS Suite_de_variables_de_creation

Modules -> ε | Global Suite_definitions Modules

Programme -> PROGRAMME PARENT_OUV Parametre_prog PARENT_FERM Corp_de_fonction

Parametre_prog -> ε | Param_prog

Param_prog -> IDENTIFIANT | IDENTIFIANT VIRGULE Param_prog

Suite_definitions : ε | Definition Suite_definitions

Definition :
ENTIER NOM_FONCTION PARENT_OUV Parametre_fonction PARENT_FERM Corp_de_fonction

Corp_de_fonction :
CREATION DEUX_POINTS Suite_de_variables_de_creation DEBUT Suite_instructions FIN
| DEBUT Suite_instructions FIN

Parametre_fonction : ε | Param_fonct

Param_fonct :IDENTIFIANT | TABLO
| IDENTIFIANT VIRGULE Param_fonct | TABLO VIRGULE Param_fonct

Suite_instructions : ε | Instruction Suite_instructions

Suite_de_variables_de_creation : ε | Suite_create

Suite_create :
IDENTIFIANT | TABLO CROCH_OUV NOMBRE CROCH_FERM
| IDENTIFIANT VIRGULE Suite_create
| TABLO CROCH_OUV NOMBRE CROCH_FERM VIRGULE Suite_create

Instruction :
IDENTIFIANT DEUX_POINTS EGALE IDENTIFIANT POINT_VIR
| IDENTIFIANT DEUX_POINTS EGALE NOMBRE POINT_VIR
| IDENTIFIANT DEUX_POINTS EGALE IDENTIFIANT PLUS NOMBRE POINT_VIR
| IDENTIFIANT DEUX_POINTS EGALE IDENTIFIANT MOINS NOMBRE POINT_VIR
| TABLO CROCH_OUV NOMBRE CROCH_FERM DEUX_POINTS EGALE IDENTIFIANT
POINT_VIR
| TABLO CROCH_OUV IDENTIFIANT CROCH_FERM DEUX_POINTS EGALE IDENTI-
FIANT POINT_VIR
| TABLO CROCH_OUV NOMBRE CROCH_FERM DEUX_POINTS EGALE NOMBRE POINT_VIR
| TABLO CROCH_OUV IDENTIFIANT CROCH_FERM DEUX_POINTS EGALE NOMBRE

POINT_VIR
| IDENTIFIANT DEUX_POINTS EGALE TABLO CROCH_OUV IDENTIFIANT CROCH_FERM
POINT_VIR
| IDENTIFIANT DEUX_POINTS EGALE TABLO CROCH_OUV NOMBRE CROCH_FERM
POINT_VIR
| IDENTIFIANT DEUX_POINTS EGALE NOM_FONCTION PARENT_OUV Parametre_fonction
PARENT_FERM POINT_VIR
| BOUCLE IDENTIFIANT FOIS Suite_instructions FIN
| HYPERBOUCLE IDENTIFIANT FOIS Suite_instructions FIN FINAL Suite_instructions FIN

ANNEXE C

L'analyse lexicale, le fichier LEX

```
%{
#include <stdlib.h>
#include "y.tab.h"
#include "table.h"
#include "outils.h"
#include "var.h"
#include "generateur.h"
#include "erreur.h"
#include "action_lex.h"
extern int yylval;
%}

blancs [ \t]+
saut [\n]
chiffre   [0-9]
nombre    {chiffre}+
lettremaj [A-Z]
lettremin [a-z]

id_fun {lettremin}+{chiffre}*
id_var {lettremaj}+{chiffre}*
id_tab {lettremaj}{lettremin}+{chiffre}*
%%
{blancs} ;
{saut}        {ligne++;}          /* un saut, donc on augmente le nombre de lignes */
#.* ;                             /* Commentaires Simple d'une seule ligne*/
"/*"          commentaire=vrai; /* Debut d'un gros commentaire */
"*/"          commentaire=faux; /* fin d'un gros commentaire */
"debut"       if (!commentaire) {ds_quoi=2;return(DEBUT);Poper(nom_fun_en_cours);}
"fin"         if (!commentaire) return(FIN);
"creation"    if (!commentaire) {action_creation();return(CREATION);}
"entier"      if (!commentaire) {ds_quoi=0;return(ENTIER);}
"boucle"      if (!commentaire) {ds_quoi=4;return(BOUCLE);}
"fois"        if (!commentaire) return(FOIS);
"hyperboucle" if (!commentaire) {ds_quoi=5;return(HYPERBOUCLE);}
"final"       if (!commentaire) return(FINAL);
"programme"   if (!commentaire) {action_programme();return(PROGRAMME);}

{id_fun} if (!commentaire) {action_fun();return(NOM_FONCTION);}
{id_var} if (!commentaire) {action_var();return(IDENTIFIANT);}
{id_tab} if (!commentaire) {action_tab();return (TABLO);}

"+" if (!commentaire) return(PLUS);
"-" if (!commentaire) return(MOINS);
"=" if (!commentaire) return(EGALE);
"(" if (!commentaire) {action_ouv();return(PARENT_OUV);}
```

```
")" if (!commentaire) {action_ferm();return(PARENT_FERM);}
"[" if (!commentaire) return(CROCH_OUV);
"]" if (!commentaire) {action_croch_ferm();return(CROCH_FERM);}
":" if (!commentaire) return(DEUX_POINTS);
";" if (!commentaire) return(POINT_VIR);
"," if (!commentaire) return(VIRGULE);

{nombre} if (!commentaire) {action_nb();return(NOMBRE);}

. if (!commentaire) {action_reste();}
%%

/* ***************  La fonction pour relancer l'analyse des fichiers ********** */
int yywrap(void)
{
 if (nb_fichier==nb_arg) return 1; /* plus de fichiers à lire */

 yyin=fopen(argument[nb_fichier++],"r"); /* réouverture du fichier */
 if (yyin==NULL)
  {
    perror("fopen");
    fprintf(stderr,"Impossible de lire: %s\n",argument[nb_fichier-1]);
    exit(0);
  }

 ds_quoi=6; /* pour l'automate */
 ligne=1;   /* on remet à jour la ligne courante */
 return 0;
}
```

ANNEXE D

Exemple complet de code assembleur

0.4.1. *La factorielle en simple boucle.*

```
0: org #30      | 12: mov 11 6  | 24: pop 0     | 36: jmp #1
1: mov 6 #0     | 13: jz 11 #17 | 25: push 6    | 37: pop 13
2: mov 7 #0     | 14: add 8 #1  | 26: jmp 0     | 38: mov 1 13
3: mov 8 #0     | 15: sub 11 #1 | 27: pop 0     | 39: int 2
4: pop 5        | 16: jmp #13   | 28: push #0   | 40: mov 4 #0
5: mov 6 #1     | 17: sub 10 #1 | 29: jmp 0     | 41: end
6: mov 7 #0     | 18: jmp #11   | 30: mov 1 #1  |
7: mov 8 #1     | 19: add 7 #1  | 31: mov 0 #2  |
8: mov 9 5      | 20: mov 6 8   | 32: mov 13 #0 |
9: jz 9 #23     | 21: sub 9 #1  | 33: pop 12    |
10: mov 10 7    | 22: jmp #9    | 34: push #37  |
11: jz 10 #19   | 23: mov 4 #0  | 35: push 12   |
```

0.4.2. *La factorielle en hyperboucle.*

```
0: org #45     | 16: push 11   | 32: add 8 #1  | 48: mov 0 #2
1: pop 6       | 17: push #21  | 33: pop 10    | 49: pop 12
2: mov 7 #0    | 18: sub 8 #1  | 34: pop 11    | 50: mov 13 #0
3: mov 5 #0    | 19: jz 8 #31  | 35: pop 7     | 51: push #54
4: mov 7 #0    | 20: jmp #10   | 36: pop 6     | 52: push 12
5: jz 6 #38    | 21: sub 11 #1 | 37: jmp 10    | 53: jmp #1
6: mov 8 6     | 22: jmp #12   | 38: mov 4 #0  | 54: pop 13
7: add 6 #1    | 23: add 8 #1  | 39: pop 0     | 55: mov 1 13
8: mov 9 6     | 24: je 9 8    | 40: push 5    | 56: int 2
9: sub 6 #1    | 25: jz 9 #38  | 41: jmp 0     | 57: mov 4 #0
10: add 7 #1   | 26: pop 10    | 42: pop 0     | 58: end
11: mov 11 7   | 27: pop 11    | 43: push #0   |
12: jz 11 #23  | 28: pop 7     | 44: jmp 0     |
13: mov 4 #0   | 29: pop 6     | 45: mov 1 #1  |
14: push 6     | 30: jmp 10    | 46: mov 0 #2  |
15: push 7     | 31: add 5 #1  | 47: mov 1 #1  |
```

On remarque que le code n'est pas beaucoup plus long et qu'à l'exécution, la rapidité comme la complexité sont presque identiques (car dans les deux cas, il est facile de voir qu'on effectue grossièrement fact(n) additions, mais le code de l'hyperboucle utilisée est un plus lent à cause de la gestion de la pile et des compteurs pour l'hyperboucle.

Bibliographie

[1] Pierre wolper. *Introduction à la Calculabilité.* InterEdition. Paris, 1991.

[2] Alain Cardon, Christian Charras. *Introduction à l'Algorithme et à la Programmation.* Ellipses. Paris, 1996.

[3] A. Finkel *Advanced Programming Language design.* Addison-Wesley

[4] Robert Mandl *On "powerloops constructs in programming languages.* Box 199 MIT Cambridge, Ma 012139 (617) 2727853 ACM SIGPLAN Notices 25(4) page 72-82 (Avril 90)

[5] R. Morales-Bueno, I. Fortes,L. Mora and F. Triguero *Two Classical Theorems Revisited.* Campus de Teatinos, 29071 Malaga, Spain

[6] N. Busi, R. Gorrieri, Gianluigi Zvattaro. Bologne Italie ; Revue : Information and Computation 156 90-121 (2000) *On the exprsivenes of Linda Coordination Primitives.*

[7] M. P. Ward *Iterative Procedures for Computing Ackerman's Function* July 16,1993

[8] W. Ackermann *Zum Hilbertschen Aufbau der reellen Zahlen* Math Ann.99 (1928), 118-133

[9] De Kerf, J. L. F. *A note on the power Operators ("Loops are Harmful")* SIGPLAN Notices, 24 (11), 102-108. 1989.

[10] A. R. Meyer, D-M Ritchie *The Complexity of loop programs* Proc. ACM Nat. Meeting 1976,465-469

[11] R. David *Un algorithme premitif récursif pour la fonction inf* Laboratoire de Mathématiques de Chambéry