Un traducteur de PL/1 vers C

M. Mazaud

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 65

UN TRADUCTEUR DE PL/1 VERS C

Monique MAZAUD

Février 1986

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. : (1) 39 63 55 11

Résumé

Nous présentons un outil d'aide à la traduction de PL/1 Multics vers C. Cet outil a été développé lors du transport sur SM90 du système SYNTAX, jusque là opérationnel sur Multics. Comme le volume de code à traduire était considérable, il nous a semblé nécessaire d'automatiser cette traduction. Le traducteur PL/1-C a été conçu de façon à ce que les programmes C produits soient syntaxiquement corrects et compilables après une intervention manuelle minimale.

Comme des différences importantes existent entre le langage source et le langage objet, un certain nombre de problèmes de traduction se sont posés ; nous donnons dans ce rapport les solutions adoptées.

Abstract

We present a tool to help the translation from Multics PL/1 to C. It was developped to facilitate the transport of the SYNTAX system from Multics to the SM90. As the amount of code to port was considerable, we needed to automatize the translation. The PL/1-C translator has been designed so that the programs produced by the translation are compiled after few manual manipulations.

As important differences between the source and the target languages exist, some problems of translation have occurred. This report describes the solutions we have chosen.

Un traducteur de PL/1 vers C

Monique Mazaud

I.N.R.I.A
Domaine de Voluceau, 78153 Le Chesnay Cedex

1. Objectifs

Le transport sur SM90 du système SYNTAX opérationnel jusque là uniquement sur Multics a été entrepris en 1985 par notre équipe en liaison avec l'Université d'Orléans. Un des choix importants à faire a été celui du langage d'implantation et notre volonté de pénétrer le monde Unix au delà de la SM90 nous a naturellement fait choisir C.

Ce transport nécessitait la réécriture en C d'un ensemble de programmes écrits initialement en PL/1. Comme le volume de code PL/1 était considérable (45 000 lignes), il nous a semblé nécessaire d'automatiser cette traduction, et ceci bien entendu en utilisant nos propres outils. Un traducteur de PL/1 en C a donc été développé. Comme cet outil devait être utilisable rapidement pour la suite du projet, il est le fruit d'un compromis entre la rapidité

d'écriture et la complexité du traducteur.

La conception du traducteur est naturellement fondée sur la comparaison entre les deux langages. Or cette comparaison ne peut aboutir sur un certain nombre de points, car il n'y pas équivalence entre les deux langages. Pour d'autres points, une équivalence pourrait être trouvée par une transformation profonde du texte initial, automatisable sans doute mais au prix d'une complication notable du traducteur. Souhaitant obtenir un outil de transport simple, nous n'avons pas implanté ces transformations coûteuses, préférant considérer ce traducteur PL/1-C comme un système d'aide à la traduction mais où l'utilisateur devrait intervenir. Les différences profondes entre les deux langages interdisant d'atteindre une traduction complète, cette attitude est acceptable, d'autant que par des diagnotics appropriés, PL/1-C aide à réaliser ces interventions manuelles.

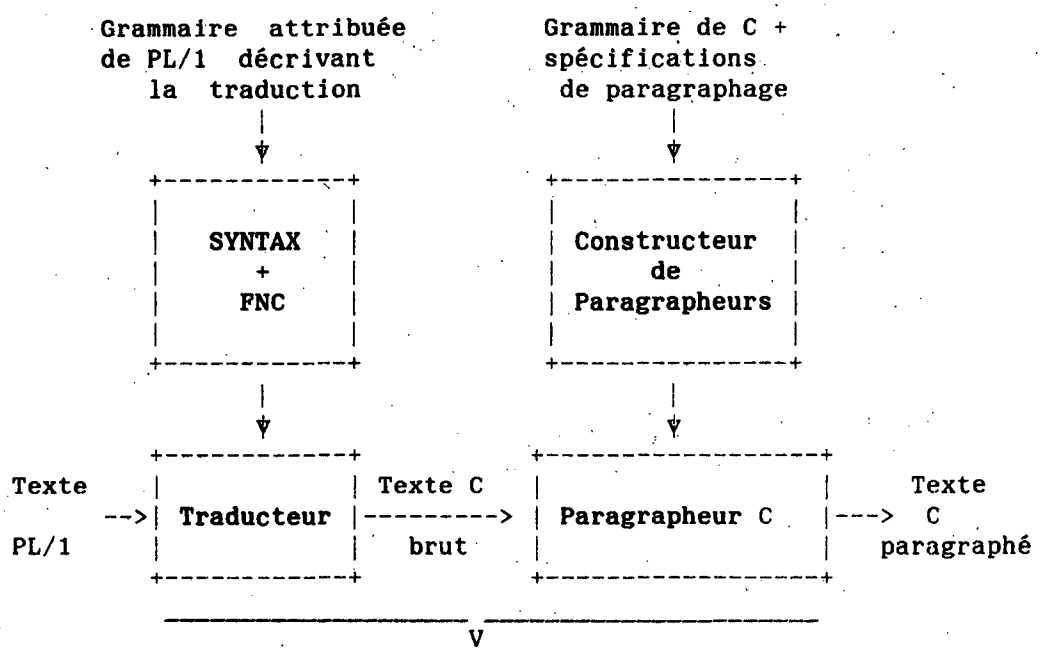
L'utilisateur peut être amené à agir sur plusieurs plans:

- sur le texte PL/1 initial pour mettre le texte sous une forme appropriée
- sur le texte C produit, avant même la compilation pour compléter la traduction.
- sur le texte C, après une première compilation, en utilisant les diagnostics du compilateur C, pour traiter des points délicats.

2. Implantation

Réalisé dans le même esprit que Pascal-Ada[1], le traducteur PL/1-C a, comme lui, pour caractéristiques principales l'efficacité, la facilité d'écriture et de maintenance. Cela a été rendu possible par l'utilisation des outils de haut niveau développés dans l'équipe (SYNTAX, FNC, Paradis).

La structure de PL/1-C est donnée par le schéma suivant:



Le traducteur est spécifié par une grammaire attribuée de PL/1. La syntaxe de PL/1 adoptée n'est pas tout à fait exhaustive par rapport à celle du PL/1 Multics de départ et ceci pour obtenir une grammaire LALR(1) (voir Annexe 1). Certains mot-clés en particulier ne sont pas acceptés (voir Annexe 2).

Parmi les attributs, on distingue essentiellement l'attribut de sortie sur le fichier objet du texte C et des attributs formant des tables internes au processus de traduction. Notons que le traducteur ne possède pas de table des symboles générale , car nous avons cherché à faire la traduction la plus simple et la plus rapide possible.

Des attributs auxiliaires permettent de traiter un certain nombre de problèmes de sémantique statique. Citons, entre autres les problèmes suivants :

- en PL/1 la syntaxe d'un appel de procédure et la syntaxe d'une référence à un élément de tableau sont identiques et ce n'est pas le cas en C. Pour distinguer entre les deux, nous avons défini un couple attribut synthétisé, attribut hérité qui contient le nom de toutes les procédures déclarées dans le programme. Cet attribut est initialisé avec les noms de toutes les fonctions incorporées PL/1. L'attribut global correspondant est hérité dans les instructions et permet d'engendrer soit un appel de procédure, soit une référence à un élément de tableau.

- en PL/1 les paramètres d'une procédure peuvent être déclarés n'importe où, avant ou après les variables globales. Le langage C impose que les paramètres soient déclarés avant l'accolade ouvrante. Pour résoudre ce problème, nous synthétisons une table des paramètres dans l'entête de la procédure et nous l'héritons dans le corps de la procédure.

- dans une déclaration PL/1, les attributs (à la PL/1) peuvent être factorisés. Les différentes informations afférant à une déclaration sont tabulées par l'intermédiaire d'un attribut. Cette table est exploitée pour produire une séquence de déclarations en C.

Le résultat de la traduction est constitué d'un ensemble de deux fichiers:

- un listage du texte PL/1 source dans lequel toutes les situations difficiles (celles sur lesquelles il faudra revenir manuellement) sont signalées par un message associé à un marqueur.

- un listage du texte C produit avec les commentaires du source PL/1, mais aussi des commentaires ajoutés par le traducteur qui indiquent en particulier à l'utilisateur où il devra intervenir.

3. Parallèlle entre PL/1 et C

La comparaison des langages PL/1 et C nous amène à distinguer trois classes de problèmes de traduction:

- celle où il y a incompatibilité entre le langage source et le langage cible .

- celle où il y a quasi-compatibilité, c'est à dire pour lesquels on peut trouver une traduction (qu'il faudra éventuellement peaufiner manuellement).

- celle où il y a équivalence , c'est à dire que la traduction est correcte .

3.1. Incompatiblité entre le langage source et le langage cible

Nous donnons ici une liste significative, mais non exhaustive des problèmes de traduction où il y a incompatibilité entre le langage source et le langage cible.

Le langage C n'admet pas, à l'encontre du langage PL/1 de blocs procéduraux. Nous aurions pu imaginer de faire faire la désimbrication des procédures PL/1 par le traducteur, mais le problème de la portée des variables n'aurait pas été résolu et nous avons préféré laisser à l'utilisateur le soin de régler ce problème lui-même. Nous donnons un exemple de transformation à réaliser au paragraphe 4.1.2.

Dans les ruptures de séquences, PL/1 autorise les références à des éléments

de tableaux d'étiquettes, or la structure de données tableau d'étiquette n'existe pas en C. Il n'y a donc pas d'équivalence possible.

Au contraire du langage C, PL/1 autorise l'utilisation de tableaux à bornes variables. Le traducteur ne résoud pas ce problème et une erreur sera diagnostiquée par le compilateur C.

Le traitement des interruptions n'est pas prévu en C, le traducteur sera dans l'impossibilité de les traduire.

3.2. Quasi-compatibilité entre le langage source et le langage cible

En C, le seul mode de passage de paramètre est le mode "valeur". Le cas des paramètres passés par référence en PL/1 doit être considéré manuellement après la traduction. Cela ne pose pas de problème particulier. En effet, il est possible de permettre à une procédure de modifier une variable dans le programme appelant. Celui-ci doit fournir l'adresse de cette variable (en utilisant un pointeur de cette variable) et la fonction appelée doit déclarer l'argument en tant que pointeur, et par son intermédiaire, désigner indirectement la vraie variable.

Pour les structures de données qui ne possèdent pas d'équivalent direct en C (chaînes de bit), nous avons défini un paquetage C qui assure l'équivalence.

La traduction des tableaux pose également un problème puisqu'en C la borne inférieure vaut toujours 0 et qu'en PL/1 celle-ci peut être quelconque. Nous avons choisi une solution de traduction correcte dans le cas où la borne inférieure est égale à 0 ou 1, ce qui représente la majorité des cas. Cette solution est incomplète et nécessite une intervention dans les autres cas.

Il existe dans PL/1 quatre attributs de "classe de mémoire" s'excluant mutuellement et indiquant la façon dont la variable doit être gérée à l'exécution. La classe par défaut est automatique, les autres sont les classes statique, pointée, contrôlée. Les variables statiques ont une existence physique pendant toute la durée du programme. Les variables automatiques ont une existence physique seulement pendant l'exécution du bloc où elles sont déclarées. Par défaut, les variables possèdent le mode automatique en PL/1 et en C. A la classe "automatic" de PL/1, on fait correspondre la classe "auto" en C. A la classe "static" de PL/1, on fait correspondre la classe "static" en C.

Les classes pointée et contrôlée devront être gérées explicitement par programme en C. Comme les variables contrôlées ne sont pas utilisées par le système SYNTAX, nous avons volontairement laissé de côté la traduction des variables possédant cette classe en PL/1. Nous montrerons au paragraphe 4.3.4.2 comment on peut gérer en C les occurrences d'une variable pointée.

3.3. Equivalence

Sur la plupart des autres aspects de la traduction, on peut dire qu'il y a équivalence et que la traduction est correcte.

4. Spécifications

4.1. Structure des programmes

Le langage C n'est pas un langage structuré en blocs comme PL/1; en effet, on ne peut définir une fonction à l'intérieur d'une autre fonction. En revanche, des variables peuvent être définies dans une structure de bloc car des déclarations peuvent suivre l'accolade gauche qui précède toute instruction composée.

4.1.1. Structure des blocs

On peut ainsi établir une correspondance entre les blocs begin ... end de PL/1 et { ... } de C.

4.1.2. Les blocs procéduraux

C n'admet pas les définitions de procédures imbriquées, aussi les sources PL/1 doivent-ils être transformés à la main pour séquentialiser les procédures des différents modules. Dans le cas où cela n'a pas été fait, un message d'erreur est produit par le traducteur:

Vérifier les imbrications de procédures

Pour des raisons techniques d'analyse syntaxique, le texte désimbriqué doit être constitué par une procédure qui englobe toutes les autres. Cette procédure peut posséder des déclarations statiques mais pas d'instructions. Le traducteur fera disparaître la procédure englobante, puisque l'imbrication n'est pas légale en C.

Pour séquentialiser un programme source PL/1, on doit désimbriquer les procédures et transformer les points d'entrée en procédures. Les règles à appliquer aux variables sont les suivantes:

- les procédures les plus imbriquées en PL/1 conservent dans la traduction C leurs variables locales.
- les variables locales des autres procédures deviennent variables globales. En effectuant ces transformations, il faut penser aux conflits de nom possibles entre ces variables, à cause des changements de portée. Considérons l'exemple suivant:

```
Main: procedure;
dcl (x2, x1) fixed bin;

p: procedure;
dcl (a, b) fixed bin;

    q1: procedure;
    dcl (u, v) fixed bin;
    ...
    ...
    end q1;

    q2: procedure;
    dcl (w, z) fixed bin;
    ...
    ...
    end q2;
...
...

end p;

end Main;
```

Le résultat de cette transformation est le suivant:

```
Main: procedure;

dcl (x2, x1) fixed bin;
dcl (a, b) fixed bin;

    q1: procedure;
    dcl (u, v) fixed bin;
    ...
    ...
    end q1;

    q2: procedure;
    dcl (w, z) fixed bin;
    ...
    ...
    end q2;

    p: procedure;
    ...
    ...
    end p;

end Main;
```

Dans cet exemple, les procédures de niveau 2 (q1, q2) sont les procédures les plus imbriquées; elles conservent leurs variables locales. Les variables

locales des procédures de niveau 1 deviennent variables globales. C'est le cas des variables locales a, b de la procédure p.

4.2. Eléments lexicaux

4.2.1. Les identificateurs

En PL/1 Multics la longueur des identificateurs est bornée par un nombre très grand (256). Sur SM90, en revanche la limitation est plus draconienne: 16 caractères(pour le compilateur C portable). Nous aurions pu décider de tronquer automatiquement les identificateurs PL/1 à 16 caractères (par une action décrite pour le constructeur lexical) , mais cela aurait donné lieu à des collisions de noms dont l'origine aurait été difficile à détecter par la suite. Nous conservons donc les identificateurs tels quels, et le diagnostic des collisions est laissé à la charge du compilateur C. L'usager pourra ainsi effectuer le renommage en toute sécurité.

En PL/1, il n'existe pas de mots réservés. On peut par exemple désigner une variable par l' identificateur "goto", or il se trouve que c'est un mot réservé de C. Dans ce cas, il y aura renommage et cela sera signalé par le système de la facon suivante:

All occurences of goto have been replaced by goto_0

si goto_0 n'existe pas par ailleurs en tant qu'identificateur dans le source PL/1 (information connue par la table des symboles de l'analyseur lexical) . Dans le cas contraire, on choisirait goto_1 etc....

4.2.2. Les chaînes de caractères

Les chaînes de caractères de PL/1 sont analysées par l'analyseur lexical de PL/1-C qui effectue les transformations dictées par les spécifications des chaînes de caractères de C.

- si la chaîne comporte un guillemet interne, il faut faire précéder ce guillemet d'un \ dans la chaine C.
- une chaîne de caractères comportant un EOL contiendra \r en C.
- un caractère de tabulation horizontale à l'intérieur d'une chaîne est traduit par \t.
- un caractère d'espacement arrière à l'intérieur d'une chaîne est traduit en \b.
- un caractère de saut de page est traduit en \f.

4.2.3. Etiquettes

C n'accepte comme étiquettes que des identificateurs. Or PL/1 offre la possibilité de travailler sur des tableaux d'étiquettes, lorsqu'on rencontrera une telle utilisation on la signalera:

```
K(1) :
| Attention: seul un identificateur est accepté comme étiquette
```

Comme K(1) n'appartient pas à la table des procédures du programme, K(1) est traduit comme un élément de tableau. De plus le même avertissement est fourni en commentaire dans le C produit, pour signaler ce problème de traduction.

4.2.4. Commentaires

Les commentaires PL/1 et C commencent par /* et se terminent par */. Par conséquent, les commentaires PL/1 sont conservés tels quels.

Les macros %include de PL/1 sont traduites en macros #include de C.

```
%include "parser_tables.en";

est traduit en:

#include "parser_tables.en"
```

4.3. Déclarations

4.3.1. Les types de données

4.3.1.1. Les variables entières

Les différentes combinaisons d'attributs, binary (bin) et fixed avec différentes qualifications donnent naissance au type int ou au type long. La règle adoptée est la suivante:

si la qualification est égale à 17 on choisit le type int, si la qualification est égale à 21 ou 35 on choisit le type long

```
dcl lgt1 bin fixed(35)      ->      long lgt1;
dcl lgt2 bin fixed(21)      ->      long lgt2;
dcl lgt3 bin fixed(17)      ->      int lgt3;
```

Dans tous les autres cas de qualification, la compatibilité du type de la variable avec un type C n'est pas prévue par le traducteur; cela est signalé à l'utilisateur.

4.3.1.2. Les chaînes

4.3.1.2.1. Les chaînes de bits

Parmi les chaînes de bits, on va distinguer les chaînes de bits de longueur 1 des autres. Les booléens n'existent pas en C. Il faut prévoir la définition d'un type boolean qui associe un entier à une chaîne de bits de longueur 1. On pourra définir par exemple, un fichier d'include qui contiendra les définitions suivantes:

```
#define boolean int
#define TRUE 1
#define FALSE 0
```

Le traducteur ayant adopté les conventions précédentes

```
dcl s_bool bit;
```

est traduit en:

```
boolean s_bool.
```

Les constantes "0"b et "1"b sont traduites respectivement par FALSE et TRUE.

Les chaînes de bits de longueur quelconque et variable ne possèdent pas d'équivalent en C, aussi un paquetage a-t-il été écrit en C pour traiter entre autres les mêmes fonctionalités qu'en PL/1. Un BITSTR est un pointeur vers une structure contenant:

- la longueur courante de la chaîne de bits;
- la longueur maximale de la chaîne, définie au moment de l'allocation
- un pointeur vers une chaîne de caractères contenant les bits; la longueur de cette chaîne est calculée en fonction de la longueur maximale de la chaîne de bits.

La spécification complète de ce type est donnée en Annexe 3.

```
dcl use bit(128);
```

est traduit en:

```
BITSTR use;
/* Attention : chaine de bits de longueur 128 */
```

```
dcl t bit(*);

est traduit en:

BITSTR t;
/* Attention : chaine de bits passée en paramètre */
```

4.3.1.2.2. Les chaînes de caractères

Une chaîne de caractères non basée est traduite en C par un tableau de caractères

```
dcl chaine char(136);

est traduit en:

char chaine[136];
```

Pour une chaîne de caractères varying un avertissement est donné pour rappeler qu'on ne sait pas traduire une chaine varying;

```
dcl xchaine char(136) varying;

est traduit en:
char xchaine[136];
/* Attention xchaine est une chaine varying dans le source PL/1 */
```

Pour un paramètre formel, la longueur du paramètre associé ne peut pas être rappelée en C.

```
dcl arg char(*) varying;

est traduit en:

char arg[];
/* Attention arg est une chaîne varying dans le source PL/1 */
```

Une chaîne de caractères basée est traduite par une déclaration de type:

```
dcl st1 char(stl1) based (st1_ptr);


est traduit en:

typedef char st1[stl1];
/* Attention: est basée sur st1_ptr */
```

4.3.1.3. Les tableaux

En C, la borne inférieure d'un tableau vaut toujours 0, or en PL/1 elle peut avoir n'importe quelle valeur (en particulier, elle peut être évaluée dynamiquement). Cependant dans SYNTAX, la borne inférieure des tableaux vaut le plus souvent 0 ou 1, on va distinguer ces cas du cas général.

4.3.1.3.1. La borne inférieure vaut 0 ou 1

Nous convenons d'associer au tableau source un tableau objet ayant les mêmes éléments, plus éventuellement un élément 0 (qui n'existe pas dans le tableau source si la borne inférieure est 1).

```
Exemple :
dcl tab(1:5) bin fixed;

est traduit en:
int tab[6];
```

Il n'est pas nécessaire de translater les indices des variables indicées de ces tableaux, car ceci n'occasionnera qu'une perte de place de un élément, dans le cas où la borne inférieure vaut 1.

4.3.1.3.2. Cas général

Dans le cas général, on engendre une déclaration de tableau avec la taille convenable et un avertissement signalant que les indices devront être translatés à la main.

```
Exemple:
dcl tab(-n:+m) bin fixed;
est traduit par

int tab[m+n+1];
```

et les indices devront être translatés de +n à la main:

Une autre solution pour la traduction des tableaux dans le cas général a été envisagée. Elle consiste à allouer un tableau intermédiaire et à définir le tableau concerné comme un pointeur sur le tableau intermédiaire en effectuant la translation nécessaire.

```
int tab_0[m+n+1];
int *tab = (&tab_0) + n;
```

Dans ce cas la référence tab(i) à un élément de tableau est traduite correctement par tab[i]. Cette solution serait valide pour les variables automatiques, mais elle serait incorrecte pour les variables statiques externes

et les paramètres. Pour des raisons de simplification de l'écriture du traducteur, nous ne l'avons pas adoptée.

Un cas particulier à considérer est celui des tableaux de chaînes de caractères. Nous distinguons le cas où un tableau de chaînes de caractères est initialisé (voir paragraphe 3.3.3) de celui où il ne l'est pas. Dans le cas général un tableau de chaînes est traduit par un tableau de tableau de caractères.

```
dcl chaines(1:2) char(6);

char chaines[3][6];
```

4.3.1.4. Les structures

Les déclarations de structures acceptées par le traducteur PL/1-C doivent être isolées des autres déclarations de variables. La résolution du cas général posait des problèmes, soit syntaxiques (trouver une grammaire LALR(1)), soit sémantiques (détection de la fin de la déclaration d'une structure). Ainsi la syntaxe du traducteur PL/1-C n'admet pas de déclarations telles que:

```
dcl x bin fixed,
    01 a,
       2 b,
         3 v bin fixed,
         3 u bin fixed,
       2 c,
    y bin fixed;
```

Chaque déclaration de structure doit être isolée et son niveau externe est imposé par la description lexicographique . Ce niveau doit être impérativement 01 (ni 1, ni 001...). La table dont la déclaration est un tableau de structures

```
dcl 01 table (1:st_top)
    , 02 code fixed bin
    , 02 lhs_nb fixed bin
    , 02 rhs_nb fixed bin
    ;
```

sera traduite en C par:

```
struct {
     int code;
     int lhs_nb;
     int rhs_nb;
       } table [1+ st_top];
```

4.3.1.5. Les pointeurs

En PL/1 un pointeur n'est pas typé par sa déclaration, par conséquent le traducteur ne possède pas d'information sur le type pointé. La convention adoptée par le traducteur est de typer systématiquement tous les pointeurs par un type entier, et de signaler à l'usager qu'il faudra corriger ensuite cette déclaration à la main.

Exemple:

```
dcl xx pointer;

est traduit par
int *xx;
/* Attention: xx est un pointeur dans le source PL/1 */
/* il faudra corriger le type du pointeur à la main */
```

4.3.1.6. Les fichiers

A l'attribut "file" de PL/1, le traducteur fait correspondre le type FILE en C. Ainsi

```
dcl CF_PARSER file internal;

est traduit par

FILE *CF_PARSER;
```

4.3.1.7. Les constantes entry

A une déclaration d'entry PL/1, on peut associer la déclaration du nom d'une procédure externe en C.

```
dcl alpha_table entry (bin fixed, ptr) returns(bin fixed);
```

est traduit en:

```
extern int alpha_table ();
```

4.3.2. L'attribut like

L'attribut like sert à abréger la déclaration d'une structure en prenant modèle, partiellement ou totalement, sur une autre. Supposons que la variable A soit déclarée en PL/1 de la facon suivante:

```
dcl 01 A,
    2 B fixed bin,
    2 C fixed bin;
```

et que la variable Z soit définie par référence à A par l'attribut like:

```
dcl 01 Z like A;
```

Comme le traducteur fonctionne sans table des symboles, la variable A ne possède pas de nom de type; nous proposons d'utiliser le nom A à cet effet:

```
struct A Z;
```

il y aura un diagnostic d'erreur du compilateur C, car ici A est en position de nom de type alors qu'il a déja été utilisé comme nom de variable. L'utilisateur devra, à la main, nommer le type de la variable A. Par exemple:

```
struct machin {int B; int C} A;
```

et utiliser ce nom de type pour la variable Z

```
struct machin Z;
```

Si l'attribut like n'est pas au niveau 01:

```
dcl 01 node based,
       02 header like node_header;

est traduit en:

typedef struct {
          node_header header;
             } node;
```

A la compilation du texte C produit:

- soit node_header est une variable pointée ,et dans ce cas elle aura été traduite par un typedef et alors tout se passera bien.
- soit elle ne l'est pas, alors node_header est une variable et il y aura un diagnostic du compilateur C. L'utilisateur devra nommer le type de la variable node_header à la main.

4.3.3. L'attribut initial

En PL/1, pour initialiser une variable on utilise l'attribut initial. La syntaxe de C consiste à fournir entre accolades la liste des valeurs initiales.

```
dcl ndigit(1:3) bin fixed init(1, 2, 3);

est traduit en:

int ndigit[4] = {0, 1, 2 , 3};
```

Le traducteur reprend simplement la liste des données de l'attribut initial,et ajoute la valeur de l'élément zéro pour un tableau dont la borne inférieure est 1.

Un tableau de chaînes de caractères initialisé est traduit par un tableau de pointeurs sur des caractères:

```
dcl chaines(1:2) char(6) initial("un", "deux");

est traduit en:

char *chaines[3] = {"", "un", "deux"};
```

4.3.4. La gestion de la mémoire

4.3.4.1. Les règles de visibilité

Un programme PL/1 est, d'une manière générale, formé de la juxtaposition d'une ou plusieurs procédures dites "externes". Le point de départ de l'éxecution est spécifié par le mot-clé "main" apparaissant dans une des instructions "procedure". Ces procédures "externes" peuvent échanger des informations de deux manières :

- par les paramètres qu'elles se passent.
- par des variables globales de type "external"

Une variable déclarée "external" ne respecte pas les règles de portée que nous avons vues plus haut. Elle peut être connue de tout le programme, mais doit apparaître dans une instruction "declare" dans chaque procédure externe où on désire la référencer. Il faut noter que "external" implique "static" (les variables doivent en effet être implantées dès le début de l'éxecution et être indépendantes des procédures). L'attribut dual de "external" est "internal" ("internal" est l'attribut par défaut).

Les règles sont identiques dans le langage C. Ainsi à l'attribut "external" de PL/1 correspond l'attribut "extern" de C.

```
dcl code fixed bin(35) external static;

est traduit par

extern static long code;
```

4.3.4.2. Les classes de mémoire

En PL/1, les blocs ne servent pas seulement à définir dans le programme des zones déterminant la portée des déclarations. Ils ont aussi un rôle relatif à l'allocation de mémoire des variables pendant l'éxécution du programme.

Ainsi que nous l'avons indiqué au paragraphe 3.2, il y a équivalence entre PL/1 et C pour les variables statiques et automatiques. Seule la traduction des variables pointées pose un problème particulier. La solution adoptée consiste à faire correspondre une déclaration de type en C à la déclaration d'une variable basée. Le nom de la variable PL/1 peut être utilisé comme nom de type, car on accède à la variable par l'intermédiaire d'un pointeur et seul le nom de ce pointeur sera utilisé par la suite.

Exemple :

texte PL/1

```
dcl 01 node_trans based,
       02 age bin fixed,
       02 texte char(15);

dcl xx pointer;

allocate node_trans set(xx);

xx -> node_trans.age = 30;
```

est traduit en :

```
typedef struct {
                int age;
                char texte[15];
               }
            node_trans;

node_trans *xx;
xx = (node_trans*) calloc(1,sizeof(node_trans));

xx -> age = 30;
```

Sur la déclaration du pointeur xx, aucun traitement n'est effectué. A l'allocation effective de la variable pointée :

```
allocate node_trans set (xx);
```

on va associer la déclaration du pointeur xx et l'allocation effective de mémoire. Une référence à la variable pointée se fait par l'intermédiaire du nom du pointeur.

4.4. Expressions

Les opérateurs sont traduits en fonction de leur correspondance en C. Cette correspondance n'existe pas pour l'opérateur d'exponentiation : alors un avertissement est engendré par le traducteur

```
Attention l'exponentiation n'est pas implantée
```

La concaténation de deux chaînes de caractères est traduite par l'appel à une procédure C appelée "catenate". (On ne peut utiliser la fonction strcat(c1, c2) de la bibliothèque C qui concatène c1 à c2 et rend le résultat dans c1).

En l'absence de table des symboles, on donne la priorité au "et logique" et au "ou logique" pour la traduction des opérateurs | et & de PL/1.

Le tableau suivant permet d'établir la correspondance entre les opérateurs PL/1 et les opérateurs C lorsqu'ils sont différents d'un langage à l'autre.

PL/1	C
||	catenate
^	~
opérateurs relationnels	
=	==
^=	!=
^<	>=
^>	<=
>=	>=
|	||
&	&&

4.5. Instructions

4.5.1. Les affectations

L'opérateur d'affectation s'écrit "=" en PL/1 et en C.

La traduction des affectations ne pose donc pas de problème particulier.

Si le source PL/1 utilise toute la puissance du langage pour les affectations, il faudra être attentif au résultat de la traduction puisque le traducteur fonctionne sans table des symboles. Par exemple, si T est déclaré comme un tableau, l'instruction source PL/1 T = 2; qui affecte 2 à tous les éléments du tableau sera traduite en C par T=2 (dans ce cas le diagnostic sera effectué par le compilateur C).

Il est possible, en PL/1 d'affecter une valeur à plusieurs variables en une seule instruction. La partie gauche de la l'affectation est alors une liste de variables séparées par des virgules. Les affectations sont effectuées de la gauche vers la droite. En C au contraire les variables à affecter sont traitées de droite à gauche, pour le reste la sémantique du langage C n'impose rien (c'est le compilateur qui décide, suivant des critères de coût en général). Nous convenons donc d'inverser la liste des variables à affecter. Cette solution doit être correcte dans la plus grande partie des cas.

Ainsi a,b = c;

est traduit par b = a = c;

4.5.2. Les instructions conditionnelles

Il y a équivalence entre

```
if <expression> then <instruction>
                [else <instruction>]
```

et

```
if ( <expression> ) <instruction>
else  <instruction>
```

Pour l'instruction conditionnelle

```
if codeˆ=0 | arg_len=0 then do;
   call com_err(code, "a name is needed");
   return;
   end;
```

le traducteur produit:

```
if (code!= 0 || arg_len==0)
   {
   com_err(code, "a name is needed");
   return;
   }
```

4.5.3. Les boucles

La traduction des boucles utilise les instructions spécifiques de C pour incrémenter ou décrémenter la variable de contrôle lorsque le pas vaut 1. Ceci est illustré par les exemples suivants:

```
do i=nbpro to 1 by -1;      for (i=nbpro; i>=1; i--) {
...                         ...
end;                        }

do i=1 to nb by 1;          for (i=1; i<=nb; i++){
...                         ...
end;                        }

do i=nbter by -1 to 1;      for (i=nbter; i>=1; i--){
...                         ...
end;                        }

do i=1 by 2 to nbter;       for (i=1; i<=nbter; i=i+2;){
...                         ...
end;                        }
```

Les boucles "while" sont traduites par des instructions "while" de C. Ainsi:

```
do while ( a < 10)
...
end;
```

est traduit par

```
while (a < 10)
{
...
}
```

4.5.4. Les appels de procédures

Un appel de procédure interne peut être reconnu par le traducteur grâce à une recherche dans une table interne (voir paragrahe 2).

Ainsi l'appel à la procèdure interne "out_const"

```
call out_const(P_mprod, "P_mprod");
```

est traduit par:

```
out_const(P_mprod, "P_mprod);
```

Si le passage de paramètre est fait par référence, il faudra revenir sur le texte C produit à la main pour simuler ce mode de passage selon la technique évoquée au paragraphe 3.2.

4.5.5. Les ruptures de séquences

Si l'étiquette référencée par le "goto" est une constante de type label, il y a équivalence entre l'instruction source et l'instruction objet.

En revanche, la référence d'un goto PL/1 à une variable label ou à un élément de tableau ou à un élément de structure, n'a pas d'équivalent en C.

Au branchement goto A(i) est associé un avertissement:

```
goto A(i)
| Attention: construction inconnue en C
```

De même, la définition d'un élément de tableau de type étiquette est signalée à l'utilisateur. Souvent l'ensemble des définitions d'un tableau d'étiquettes peut être traduite par un "switch". Mais il faut être vigilant car il peut y avoir un goto calculé à l'intérieur de la définition d'une variable indicée de type étiquette.

4.5.6. Les instructions d'entrées-sorties

Les instructions d'entrée/sortie n'ont pas été implantées pour des raisons de commodité ; le but étant d'obtenir rapidement un traducteur opérationnel dans le contexte du transport de SYNTAX sur SM90.

Le texte de chaque instruction d'entrée/sortie est mis en commentaire d'un avertissement à l'utilisateur.

Ainsi l'instruction PL/1:

```
put file(F_listing) skip(5);
```

est traduite par:

```
/* Attention: les opérations d'entrée/sortie
   ne sont pas implantées put file(F_listing) skip(5) */
```

5. Résultats

Ce traducteur s'est avéré être un outil efficace d'aide à la traduction des programmes de SYNTAX écrits en PL/1 vers C.

Réalisé en 4 mois, et utilisé pour traduire 45 000 lignes de source PL/1, il a grandement simplifié la tâche de transport de SYNTAX sur SM90. Les auteurs du transport ont été guidés dans leur tâche de réécriture quand celle-ci n'avait pas été faite automatiquement, ils ont pu s'attacher plus rapidement à résoudre les problèmes d'environnement avec le système.

On peut donner une idée du gain de temps réalisé en citant l'exemple du programme "TABACT". A partir du programme source en PL/1, il a fallu environ 5 journées de travail pour obtenir à l'aide du traducteur PL/1-C 5000 lignes de texte C compilables.

Ajoutons également que PL/1-C a servi à traduire en C un système de manipulation d'arbres.

C'est un outil perfectible. Des améliorations notables pourraient lui être apportées. Citons entre autres:

- la désimbrication automatique des procédures PL/1 conformément aux règles énoncées au paragraphe 4.1.2.

- la traduction des instructions d'entrées-sorties. Cette traduction nécessite un volume de codage assez important mais elle ne pose pas de problème particulier.

Nous avons dû faire des choix afin d'obtenir rapidement un outil opérationnel et donc limiter la traduction. Mais la nature des outils utilisés permettrait d'étendre largement le sous-ensemble de PL/1 accepté. Plus généralement, en dépassant le cas des langages PL/1 et C, il nous apparaît que des outils de haut niveau comme SYNTAX, FNC, Paradis sont essentiels pour réaliser des transformations entre langages par l'aide efficace qu'ils apportent et le gain considérable en temps qui en est retiré.

ANNEXE1

Syntaxe du PL/1 accepté par le traducteur PL/1-C

```
<AXIOME>          = <PROC> ";"                                    ;
<AXIOME>          = <PREFEL> <PROC> ";"                           ;
<PROC>            = <PROCHEAD> ";" <P-CORPS>                      ;
<P-CORPS>         = <SENTL> <ENDCLAUSE>                           ;
<P-CORPS>         = <ENDCLAUSE>                                   ;
*
*           DEFINITIONS DE TETE DE PROCEDURE
*
<PROCHEAD>        = <LABELL> procedure <PATTRIS>                  ;
<PROCHEAD>        = <LABELL> proc <PATTRIS>                       ;
<PATTRIS>         = <OPTIONS>                                     ;
<PATTRIS>         = <PARAML> <OPTIONS>                            ;
<PATTRIS>         = <PARAML> <PROCOPTL> <OPTIONS>                 ;
<PATTRIS>         = <PROCOPTL> <OPTIONS>                          ;
<OPTIONS>         =                                               ;
<OPTIONS>         = options ( <OPTL> )                            ;
<PROCOPTL>        = <REC> <RET>                                   ;
<PROCOPTL>        = <RET> <REC>                                   ;
<PROCOPTL>        = <RET>                                         ;
<PROCOPTL>        = <REC>                                         ;
<PARAML>          = ( <IDCOMMAL> )                                ;
<PARAML>          = ( )                                           ;
<REC>             = recursive                                     ;
<RET>             = returns ( <FUNCATT> )                         ;
*
*           FIN DE PROCEDURE, BLOC ...
<ENDCLAUSE>       = end                                           ;
<ENDCLAUSE>       = <LABELL> end                                  ;
<ENDCLAUSE>       = end <ID>                                      ;
<ENDCLAUSE>       = <LABELL> end <ID>                             ;
*
*           LABELS
<LABELL>          = <LABELL> <LABEL>                              ;
<LABELL>          = <LABEL>                                       ;
<LABEL>           = <ID> :                                        ;
<LABEL>           = <REFIND> :                                    ;
*
*           PREFIXES CONDITION
<PREFEL>          = <PREFEL> <PREFPAR>                            ;
<PREFEL>          = <PREFPAR>                                     ;
<PREFPAR>         = ( <PREFS> )                                   ;
<PREFS>           = <PREFS> , <PREF>                              ;
<PREFS>           = <PREF>                                        ;
<PREF>            = check ( <IDCOMMAL1> )                         ;
<PREF>            = nocheck ( <IDCOMMAL1> )                       ;
<PREF>            = <ID>                                          ;
```

```
*
*
*               DECLARATIONS
*
<NONEXEC>               = <PROC>                                    ;
<NONEXEC>               = <ENTRY>                                   ;
<NONEXEC>               = <FORMAT>                                  ;
<NONEXEC>               = <DECLARE>                                 ;
*
*               FORMATS
*
<FORMAT>                = <LABELL> format ( <SPECFORMAT> )          ;
<SPECFORMAT>            = <SPECFORMAT> , <SPEC>                     ;
<SPECFORMAT>            = <SPEC>                                    ;
<SPEC>                  = <ATOMIC_SPEC>                             ;
<SPEC>                  = %ENTIER <BASIC_SPEC>                      ;
<SPEC>                  = <EXPAR> <BASIC_SPEC>                      ;
<BASIC_SPEC>            = ( <SPECFORMAT> )                          ;
<BASIC_SPEC>            = <ATOMIC_SPEC>                             ;
<ATOMIC_SPEC>           = <ID>                                      ;
<ATOMIC_SPEC>           = <REFIND>                                  ;
*
*               ENTRY ET RETURN
*
<ENTRY>                 = <LABELL> entry <PATTRIS>                  ;
<RETURN>                = return                                    ;
<RETURN>                = return ( <EXP> )                          ;
*
*               STRUCTURE GENERALE: LA ZONE DE CODE
*
<SENTL>                 = <SENTL> <SENT>                            ;
<SENTL>                 = <SENT>                                    ;
<SENT>                  = <STAT>                                    ;
<SENT>                  = <PREFEL> <STAT>                           ;
<STAT>                  = <NONEXEC> ";"                             ;
<STAT>                  = <UCSTAT> ";"                              ;
<STAT>                  = <CSTAT>                                   ;
*
*               INSTRUCTIONS: PRETYPE DEFINITIONS
*
<UCSTAT>                = <BEGIN>                                   ;
<UCSTAT>                = <LABELL> <BEGIN>                          ;
<UCSTAT>                = <DOGROUP>                                 ;
<UCSTAT>                = <LABELL> <DOGROUP>                        ;
<UCSTAT>                = <STMT>                                    ;
<UCSTAT>                = <LABELL> <STMT>                           ;
*
```

```
*               LES INSTRUCTIONS: L'INSTRUCTION IF
*
<CSTAT>                 = <IF> <SENT>                                   ;
<CSTAT>                 = <IF> </STAT/> else <SENT>                     ;
<IF>                    = if <COND> then                                ;
<IF>                    = <LABELL> if <COND> then                       ;
</STAT/>                = <UCSTAT> ";"                                  ;
</STAT/>                = <PREFEL> <UCSTAT> ";"                         ;
</STAT/>                = <IF> </STAT/> else </STAT/>                   ;
</STAT/>                = <PREFEL> <IF> </STAT/> else </STAT/>          ;
*
*               LES INSTRUCTIONS:
*
*               LES STRUCTURES DE BLOC: LE BLOCK BEGIN
*
<BEGIN>                 = begin ";" <G-CORPS>                           ;
*
*               LES STRUCTURES DE BLOCK: LE BLOCK DO
*
<DOGROUP>               = <DOHEAD> ";" <G-CORPS>                        ;
<DOHEAD>                = <NON_ITER_DO>                                 ;
<DOHEAD>                = <ITER_DO>                                     ;
<NON_ITER_DO>           = do                                            ;
<ITER_DO>               = <DO_WHILE>                                    ;
<ITER_DO>               = <MULTIPLE_DO>                                 ;
<DO_WHILE>              = do while ( <COND> )                           ;
<MULTIPLE_DO>           = do <VARITER>                                  ;
<VARITER>               = <REFERENCE> = <CONTROL>                       ;
<VARITER>               = <VARITER> , <CONTROL>                         ;
<CONTROL>               = <SINGLE_LOOP>                                 ;
<CONTROL>               = <SINGLE_LOOP> while ( <COND> )                ;
<CONTROL>               = <REPEAT_CONTROL>                              ;
<CONTROL>               = <REPEAT_CONTROL> while ( <COND> )             ;
<CONTROL>               = <FORTRAN_CONTROL>                             ;
<CONTROL>               = <FORTRAN_CONTROL> while ( <COND> )            ;
<SINGLE_LOOP>           = <EXP>                                         ;
<REPEAT_CONTROL>        = <EXP> repeat <EXP>                            ;
<FORTRAN_CONTROL>       = <EXP> to <EXP>                                ;
<FORTRAN_CONTROL>       = <EXP> to <EXP> by <EXP>                       ;
<FORTRAN_CONTROL>       = <EXP> by <EXP>                                ;
<FORTRAN_CONTROL>       = <EXP> by <EXP> to <EXP>                       ;
*
*               CORPS ET FIN DE DO, BEGIN
*
<G-CORPS>               = <SENTL> <ENDCLAUSE>                           ;
<G-CORPS>               = <ENDCLAUSE>                                   ;
*
```

```
*
*                LES INSTRUCTIONS SIMPLES: DEFINITION
*
<STMT>                  = <CALL>                                    ;
<STMT>                  = <RETURN>                                  ;
<STMT>                  = <STOP>                                    ;
<STMT>                  = <ASSIGN>                                  ;
<STMT>                  = <I/O>                                     ;
<STMT>                  = <ON>                                      ;
<STMT>                  = <REVERT>                                  ;
<STMT>                  = <SIGNAL>                                  ;
<STMT>                  = <ALLOCATE>                                ;
<STMT>                  = <FREE>                                    ;
<STMT>                  = <DEBUG>                                   ;
<STMT>                  = <NUL>                                     ;
<STMT>                  = <GOTO>                                    ;
*
*               LES INSTRUCTIONS SIMPLES
*
*               1 L'INSTRUCTION NULLE
*
<NUL>                   =                                           ;
*
*               2 L'INSTRUCTION APPEL DE PROCEDURE
<CALL>                  = call <REFERENCE>                          ;
*
*               3 L'INSTRUCTION D'ARRET
*
<STOP>                  = stop                                      ;
*
*               4 L'INSTRUCTION d'AFFECTATION
<ASSIGN>                = <REFL> = <EXP>                            ;
<REFL>                  = <REFERENCE>                               ;
<REFL>                  = <REFL> , <REFERENCE>                      ;
*
*               5 LES UNITES DE CONDITION
<ON>                    = on <NOMCOND> snap <UNITE>                 ;
<ON>                    = on <NOMCOND> <UNITE>                      ;
<REVERT>                = revert <NOMCOND>                          ;
<SIGNAL>                = signal <NOMCOND>                          ;
<NOMCOND>               = <ID>                                      ;
<NOMCOND>               = <ID> ( <REFERENCE> )                      ;
<UNITE>                 = system                                    ;
<UNITE>                 = <BEGIN>                                   ;
<UNITE>                 = <STMT>                                    ;
*
```

```
*               6 LES DEUX INSTRUCTIONS D'ALLOCATION DYNAMIQUE DES VARIABLES
*
<ALLOCATE>              = allocate <AIDL>                                   ;
<ALLOCATE>              = alloc <AIDL>                                      ;
<AIDL>                  = <AIDL> , <AID>                                    ;
<AIDL>                  = <AID>                                             ;
<AID>                   = <ID>                                              ;
<AID>                   = <ID> set ( <REFERENCE> )                          ;
<AID>                   = <ID> set ( <REFERENCE> ) in ( <REFERENCE> )       ;
<FREE>                  = free <FIDL>                                       ;
<FIDL>                  = <FIDL> , <FID>                                    ;
<FIDL>                  = <FID>                                             ;
<FID>                   = <REFERENCE>                                       ;
<FID>                   = <REFERENCE> in ( <REFERNCE> )                     ;
*
*               7 LA RUPTURE DE SEQUENCE
*
<GOTO>                  = goto <REFERENCE>                                  ;
<GOTO>                  = go to <REFERENCE>                                 ;
*
*               8 LES INSTRUCTIONS DE MISE AU POINT
*
<DEBUG>                 = check                                             ;
<DEBUG>                 = flow                                              ;
<DEBUG>                 = nocheck                                           ;
<DEBUG>                 = noflow                                            ;
*
*               9 LES INSTRUCTIONS D'ENTREE SORTIE
<I/O>                   = read <IDFILE> <RSPEC>                             ;
<I/O>                   = write <IDFILE> <WSPEC>                            ;
<I/O>                   = rewrite <IDFILE> <WSPEC>                          ;
<I/O>                   = delete <IDFILE>                                   ;
<I/O>                   = delete <IDFILE> key ( <EXP> )                     ;
<I/O>                   = get <SPECIF> <GDATAL>                             ;
<I/O>                   = get <SPECIF1>                                     ;
<I/O>                   = put <SPECIF> <PDATAL>                             ;
<I/O>                   = put <SPECIF1>                                     ;
<I/O>                   = open <OSPECIF>                                    ;
<I/O>                   = close <IDFILE_LIST>                               ;
<GDATAL>                = <GDATA>                                           ;
<SPECIF>                =                                                   ;
<SPECIF1>               = <IDFILE>                                          ;
<SPECIF1>               = <SPECIAL>                                         ;
<SPECIF1>               = <IDFILE> <SPECIAL>                                ;
<SPECIF1>               = string ( <REFERENCE> )                            ;
<IDFILE>                = file ( <FILENAME> )                               ;
<IDFILE_LIST>           = <IDFILE>                                          ;
<IDFILE_LIST>           = <IDFILE_LIST> , <IDFILE>                          ;
<FILENAME>              = <REFERENCE>                                       ;
<SPECIAL>               = skip                                              ;
<SPECIAL>               = skip  ( <EXP> )                                   ;
<SPECIAL>               = page                                              ;
<SPECIAL>               = page ( <EXP> )                                    ;
```

```
<PDATAL>              = list ( <PEXPL> )                          ;
<PDATAL>              = edit ( <PEXPL> ) ( <FORMATL>              ;
<PDATAL>              = data ( <GREFEL> )                         ;
<PEXPL>               = <PEXPL> , <PEXP>                          ;
<PEXPL>               = <PEXP>                                    ;
<PEXP>                = ( <PEXPL> do <VARITER> )                  ;
<PEXP>                = <EXP>                                     ;
<GDATA>               = list ( <GREFEL> )                         ;
<GDATA>               = edit ( <GREFEL> ) ( <FORMATL> )           ;
<GDATA>               = data ( <GREFEL> )                         ;
<GREFL>               = <GREFL> , <GREF>                          ;
<GREFL>               = <GREF>                                    ;
<GREF>                = <REFERENCE>                               ;
<GREFL>               = ( <GREFL> do <VARITER> )                  ;
<FORMATL>             = <SPECFORMAT>                              ;
<RSPEC>               = into ( <BUF> )                            ;
<RSPEC>               = into ( <BUF> ) key ( <EXP> )              ;
<WSPEC>               = from ( <BUF> )                            ;
<WSPEC>               = from ( <BUF> ) key ( <EXP> )              ;
<OSPECIF>             = <OSPECIF> , <OSPEC>                       ;
<OSPECIF>             = <OSPEC>                                   ;
<OPSEC>               = <IDFILE> <SPECO_LIST>                     ;
<SPECO_LIST>          = <SPECO>                                   ;
<SPECO_LIST>          = <SPECO_LIST> <SPECO>                      ;
<SPECO>               = <ID>                                      ;
<SPECO>               = environment ( <ID> )                      ;
<SPECO>               = <ID> ( <EXP> )                            ;
<BUF>                 = <REFERENCE>                               ;
*
*              LISTES de PARAMETRES
*
<IDCOMMAL>            = <IDCOMMAL> , <ID>                         ;
<IDCOMMAL>            = <ID>                                      ;
<IDCOMMAL1>           = <IDCOMMAL1> , <ID>                        ;
<IDCOMMAL1>           = <ID>                                      ;
<OPTL>                = <OPTL> , <ID>                             ;
<OPTL>                = <ID>                                      ;
*
*              EXPRESSIONS ET CONDITIONS
*
<COND>                = <EXP>                                     ;
<EXP*>                = <EXP>                                     ;
<EXP*>                = *                                         ;
<EXP>                 = <EXP7>                                    ;
<EXP>                 = <EXP> | <EXP7>                            ;
<EXP7>                = <EXP6>                                    ;
<EXP7>                = <EXP7> & <EXP6>                           ;
<EXP6>                = <EXP5>                                    ;
<EXP6>                = <EXP6> <C> <EXP5>                         ;
<EXP5>                = <EXP4>                                    ;
<EXP5>                = <EXP5> || <EXP4>                          ;
<EXP4>                = <EXP3>                                    ;
<EXP4>                = <EXP4> + <EXP3>                           ;
<EXP4>                = <EXP4> - <EXP3>                           ;
```

```
<EXP3>                  = <EXP2>                                    ;
<EXP3>                  = <EXP3> * <EXP2>                           ;
<EXP3>                  = <EXP3> / <EXP2>                           ;
<EXP2>                  = <EXP1>                                    ;
<EXP2>                  = <EXP1> ** <EXP2>                          ;
<EXP2>                  = + <EXP2>                                  ;
<EXP2>                  = - <EXP2>                                  ;
<EXP2>                  = ^ <EXP2>                                  ;
<EXP1>                  = <EXPPAR>                                  ;
<EXP1>                  = <REFERENCE>                               ;
<EXP1>                  = <CONST>                                   ;
<EXPPAR>                = ( <EXP> )                                 ;
<C>                     = "<"                                       ;
<C>                     = "<="                                      ;
<C>                     = "="                                       ;
<C>                     = ">="                                      ;
<C>                     = ">"                                       ;
<C>                     = "^<"                                      ;
<C>                     = "^="                                      ;
<C>                     = "^>"                                      ;
*
*               DEFINITIONS DE VARIABLES
*
<DECLARE>               = declare <DECL>                            ;
<DECLARE>               = dcl <DECL>                                ;
<DECL>                  = <DECL_STRUCTURE>                          ;
<DECL>                  = <DECL_NON_STRUCTUREE>                     ;
<DEC>                   = <DEC3>                                    ;
<DECL_NON_STRUCTUREE>   = <DECL_NON_STRUCTUREE> , <DEC3>            ;
<DECL_NON_STRUCTUREE>   = <DEC3>                                    ;
<DECL_STRUCTURE>        = %ZERO_UN <DEC3> , <LDECNUM>               ;
<DECL_STRUCTURE>        = %ZERO_UN <DEC3>                           ;
<LDECNUM>               = <LDECNUM> , %ENTIER <DEC3>                ;
<LDECNUM>               = %ENTIER <DEC3>                            ;
<DEC3>                  = <DEC2> <ATTLO>                            ;
<DEC2>                  = <DEC1>                                    ;
<DEC2>                  = <DEC1> <DIMA>                             ;
<DEC1>                  = <ID>                                      ;
<DEC1>                  = ( <DEC3L> )                               ;
<DEC3L>                 = <DEC3>                                    ;
<DEC3L>                 = <DEC3L> , <DEC3>                          ;
<DIMA>                  = ( <BPL> )                                 ;
<BPL>                   = <BP>                                      ;
<BPL>                   = <BPL> , <BP>                              ;
<BP>                    = <EXP*>                                    ;
<BP>                    = <EXP> : <EXP>                             ;
*
```

```
*                LES DIFFERENTS ATTRIBUTS DES VARIABLES
*
<ATTLO>                 = <ATTLO> <ATTO>                    ;
<ATTLO>                 =                                   ;
<ATTO>                  = <ATT>                             ;
<ATT>                   = initial <INIT>                    ;
<ATT>                   = init <INIT>                       ;
<ATT>                   = environment ( <ENVL> )            ;
<ATT>                   = entry                             ;
<ATT>                   = entry ( <DESC> )                  ;
<ATT>                   = <RET>                             ;
<ATT>                   = picture <PIC-CHAINE>              ;
<ATT>                   = pic <PIC-CHAINE>                  ;
<ATT>                   = <ID>                              ;
<ATT>                   = <ID> ( <QUAL> )                   ;
<QUAL>                  = <EXP*>                            ;
<QUAL>                  = <EXP*> , %ENTIER                  ;
<INIT>                  = ( <IVI_LIST> )                    ;
<IVI_LIST>              = <IVI_LIST> , <IVI>                ;
<IVI_LIST>              = <IVI>                             ;
<IVI>                   = <EXPAR> <INIT>                    ;
<IVI>                   = <IVAL>                            ;
<IVI>                   = <EXPAR> <IVAL>                    ;
<IVI>                   = <EXPAR>                           ;
<IVAL>                  = <REF-OR-CONST>                    ;
<IVAL>                  = + <REF-OR-CONST>                  ;
<IVAL>                  = - <REF-OR-CONST>                  ;
<IVAL>                  = ^ <REF-OR-CONST>                  ;
<IVAL>                  = *                                 ;
<REF-OR-CONST>          = <REFERENCE>                       ;
<REF-OR-CONST>          = %ENTIER                           ;
<REF-OR-CONST>          = %REEL                             ;
<REF-OR-CONST>          = %IMAG                             ;
<REF-OR-CONST>          = %REEL + %IMAG                     ;
<REF-OR-CONST>          = %REEL - %IMAG                     ;
<REF-OR-CONST>          = %CHAINE                           ;
<REF-OR-CONST>          = %CHBIT                            ;
<ENVL>                  = <ENVL> , <ENV>                    ;
<ENVL>                  = <ENV>                             ;
<ENV>                   = <ID>
<ENV>                   = ( <ID> , <DEF> )                  ;
<DEF>                   = <ID>                              ;
<DEF>                   = %ENTIER                           ;
<DESC>                  = <DESC> , <DES>                    ;
<DES>                   = <DES>                             ;
<DES>                   = %ENTIER <DESC3>                   ;
<DES>                   = <DESC3>                           ;
<DESC3>                 = <DIMA> <ATTLO>                    ;
<DESC3>                 = <ATTLO>                           ;
<FUNCATT>               = <DEC>                             ;
*
```

```
*                REFERENCE AUX VARIABLES
*
<REFERENCE>              = <POINTEUR> <REFQUAL>                ;
<REFERENCE>              = <REFQUAL>                           ;
<POINTEUR>               = <REFERENCE> ->                      ;
<REFQUAL>                = <REFQUAL> . <NAME>                  ;
<REFQUAL>                = <NAME>                              ;
<NAME>                   = <REFIND> ( <LISTIND> )              ;
<NAME>                   = <REFIND> ( )                        ;
<NAME>                   = <ID> ( )                            ;
<NAME>                   = <REFIND>                            ;
<NAME>                   = <ID>                                ;
<REFIND>                 = <ID> ( <LISTIND> )                  ;
<LISTIND>                = <LISTIND> , <EXP*>                  ;
<LISTIND>                = <EXP*>                              ;
*
*                CONSTANTES
*
<PIC-CHAINE>             = %CHAINE                             ;
<CONST>                  = %REEL                               ;
<CONST>                  = %IMAG                               ;
<CONST>                  = %ENTIER                             ;
<CONST>                  = %CHAINE                             ;
<CONST>                  = %CHBIT                              ;
<ID>                     = allocate                            ;
<ID>                     = alloc                               ;
<ID>                     = begin                               ;
<ID>                     = by                                  ;
<ID>                     = call                                ;
<ID>                     = check                               ;
<ID>                     = close                               ;
<ID>                     = data                                ;
<ID>                     = delete                              ;
<ID>                     = do                                  ;
<ID>                     = edit                                ;
<ID>                     = end                                 ;
<ID>                     = file                                ;
<ID>                     = flow                                ;
<ID>                     = free                                ;
<ID>                     = from                                ;
<ID>                     = get                                 ;
<ID>                     = go                                  ;
<ID>                     = goto                                ;
<ID>                     = in                                  ;
<ID>                     = into                                ;
<ID>                     = key                                 ;
```

```
<ID>                    = list                  ;
<ID>                    = nocheck               ;
<ID>                    = noflow                ;
<ID>                    = on                    ;
<ID>                    = open                  ;
<ID>                    = options               ;
<ID>                    = page                  ;
<ID>                    = picture               ;
<ID>                    = pic                   ;
<ID>                    = put                   ;
<ID>                    = read                  ;
<ID>                    = recursive             ;
<ID>                    = repeat                ;
<ID>                    = revert                ;
<ID>                    = rewrite               ;
<ID>                    = set                   ;
<ID>                    = signal                ;
<ID>                    = skip                  ;
<ID>                    = stop                  ;
<ID>                    = string                ;
<ID>                    = system                ;
<ID>                    = then                  ;
<ID>                    = to                    ;
<ID>                    = write                 ;
```

ANNEXE2

Mots clés qui ne sont pas acceptés comme identificateurs par le traducteur

else
entry
environment
format
if
initial
init
return
returns
snap
while

ANNEXE3

Le paquetage BITSTR

1. Noms

bitalloc, bitcalloc, bitfree, bittaballoc, bittabcalloc, bittabfree, bitlength, bitor, bitand, bitminus, bitnot, setlbit, resetlbit, assignlbit, bitset, bitreset, bitcopy, bitsubstr, bitassign, bitconcat, bitequal, bittest, bittrue, bitindex0, bitindex1, set_cardinal, set_get_first_elem, set_get_next_elem, bitscan, bitprint, bitstrread, bitstrget, bitstrwrite - traitement de chaînes de bits de longueur quelconque

2. Profils et description des différentes procédures

- BITSTR bitalloc (length)
  int length;

  bitalloc: alloue une chaîne de bits de longueur maximale length et de longueur courante nulle, et rend son adresse (c'est-à-dire un BITSTR); définie par macro, donc non redéfinissable.

- BITSTR bitcalloc (length)
  int length;

  bitcalloc alloue une chaîne de bits de longueur maximale length et de longueur courante length remplie de zéros, et rend son adresse.

- bitfree (p)
  BITSTR *p;

  bitfree libère la place occupée par une chaîne de bits;
  p est l'adresse du pointeur vers la chaîne (BITSTR);
  ce pointeur est remis a NULL.

- BITSTR *bittaballoc (binf, bsup, length)
  int binf, bsup, length;

  bittaballoc alloue un tableau de chaînes de bits dont les bornes sont binf et bsup ;la longueur maximale des chaînes est length et leur longueur courante est nulle; définie par macro.

- BITSTR *bittabcalloc (binf, bsup, length)
  int binf, bsup, length;

bittabcalloc alloue un tableau de chaînes de bits dont les bornes sont binf et bsup ; la longueur maximale des chaînes et leur longueur courante est length , et elles sont remplies de zéros.

- bittabfree (t, binf)
  BITSTR **t;
  int binf;

  bittabfree libère la place occupée par le tableau de chaînes de bits dont l'adresse est contenue dans t et dont la borne inférieure est binf; le pointeur vers le tableau (*t) est remis à NULL.

- int bitlength (s)
  BITSTR s;

  bitlength rend la longueur courante de la chaîne de bits s ; définie par macro.

- bitor (d, s1, s2)
  BITSTR d, s1, s2;

  bitor met dans l'union ensembliste (OU logique) de s1 et s2 ; la chaîne la plus courte est (virtuellement, comme pour les autres fonctions) complétée par des zéros.

- bitand (d, s1, s2)
  BITSTR d, s1, s2;

  bitand met dans d l'intersection ensembliste (ET logique) de s1 et s2 ; la chaîne la plus courte est complétée par des zéros.

- bitminus (d, s1, s2)
  BITSTR d, s1, s2;

  bitminus met dans d la différence ensembliste de s1 et s2 ; la chaîne la plus courte est complétée par des zéros.

- bitnot (d, s)
  BITSTR d, s;

  bitnot met dans d le complément logique de s .

- setibit (s, i)
  BITSTR s;
  int i;

setlbit met à 1 le i -ème bit de s ; setlbit n'a pas d'effet si i est en dehors des bornes (<= 0 ou > longueur courante).

- resetlbit (s, i)
  BITSTR s;
  int i;

  resetlbit met à 0 le i -ème bit de s ; n'a pas d'effet si i est en dehors des bornes.

- assignlbit (s, i, v)
  BITSTR s;
  int i, v;

  assignlbit v dans le i -ème bit de s ;selon les conventions habituelles du langage C, v est considéré comme valant 1 s'il est différent de 0; n'a pas d'effet si i est en dehors des bornes.

- bitset (s)
  BITSTR s;

  bitset remplit s de 1 sans modifier sa longueur courante.

- bitreset (s)
  BITSTR s;

  bitreset remplit s de 0 sans modifier sa longueur courante.

- bitcopy (d, s)
  BITSTR d, s;

  bitcopy copie s dans d avec troncature éventuelle si d est trop petit.

- bitsubstr (d, s, deb, length)
  BITSTR d, s;
  int deb, length;

  bitsubstr met dans d la sous-chaîne de s de début deb et de longueur length; tronque si cette sous-chaîne déborde de s, et tronque si length est supérieur à la longueur maximale de d .

- bitassign (d, deb, length, s)
  BITSTR d, s;
  int deb, length;

  bitassign remplace par s la sous-chaîne de d de début deb et de longueur length ; cette sous-chaîne est limitée par la longueur courante de s .

- bitconcat (d, s1, s2)
  BITSTR d, s1, s2;

  bitconcat place dans d la concaténation de s1 et s2 .

- int bitequal (s1, s2)
  BITSTR s1, s2;

  bitequal rend vrai (donc 1) si s1 et s2 sont égales après prolongement de la plus courte par des zéros.

- int bittest (s, i)
  BITSTR s;
  int i;

  bittest rend la valeur du i -ème bit de s ; rend -1 si i est en dehors des bornes.

- int bittrue (s)
  BITSTR s;

  bittrue rend vrai si au moins un bit de s n'est pas nul.

- int bitindex0 (s)
  BITSTR s;

  bitindex0 rend l'index dans s du premier bit nul, ou 0 si aucun bit n'est nul.

- int bitindex1 (s)
  BITSTR s;

  bitindex1 rend l'index dans s du premier bit non nul, ou 0 si tous les bits sont nuls; définie par macro.

- int set_cardinal (s)
  BITSTR s;

  set_cardinal rend le cardinal de l'ensemble représenté par s , c'est-à-dire le nombre de bits égaux à 1.

- int set_get_first_elem (s)
  BITSTR s;

set_get_first_elem rend l'index dans s du premier bit non nul, ou 0 si tous les bits sont nuls; set_get_next_elem rend l'index dans s du premier bit non nul strictement après i ,
ou 0 si tous ces bits sont nuls; ces deux fonctions sont la plupart du temps employées ensemble pour faire des boucles sur tous les bits non nuls d'une chaîne de bits.
set_get_first_elemest définie par macro.

- int set_get_next_elem (s, i)
  BITSTR s;
  int i;

- bitscan (s)
  BITSTR s;

  bitscan lit sur l'entrée standard une chaîne de bits (sous forme imprimable) et la met dans s ; la lecture s'arrête dès qu'on rencontre un caractère différent de `0' et de `1'; la longueur courante de s devient le minimum de sa longueur maximale et de la longueur de la chaîne lue.

- bitprint (s)
  BITSTR s;

  bitprint imprime sur la sortie standard la chaîne de bits s suivie d'une fin de ligne.n

- int bitstrread (f, s)
  int f;
  BITSTR s;

  bitstrread lit dans le fichier f une chaîne de bits (sous forme interne) et la met dans s ; f doit être un descripteur de fichier ouvert par open (2) et s une chaîne de bits préalablement allouée; rend le nombre d'octets lus.

- BITSTR bitstrget (f)
  int f;

  bitstrget lit dans le fichier f une chaîne de bits (sous forme interne) et lui alloue juste la place nécessaire; f doit être un descripteur de fichier ouvert par open (2); bitstrget rend le BITSTR pointant sur cette chaîne.

- int bitstrwrite (f, s)
  int f;
  BITSTR s;

  bitstrwrite " écrit" la forme interne de la chaîne de bits s dans le fichier f (ouvert par open (2) ou par create (2));rend le nombre d'octets

3. DESCRIPTION

Bitstr est un paquetage de traitement de chaînes de bits de longueur quelconque et variable. Un BITSTR est un pointeur vers une structure contenant:

- la longueur courante de la chaîne de bits;
- la longueur maximale de la chaîne, définie au moment de l'allocation;
- un pointeur vers une chaîne de caractères (allouée elle aussi dynamiquement) contenant les bits; la longueur de cette chaîne est calculée en fonction de la longueur maximale de la chaîne de bits.

Le type d'une chaîne de bits est BITSTR .Donc, pour déclarer (avant d'allouer) une chaîne de bits il faut écrire: BITSTR s;

L'allocation est laissée aux soins de l'utilisateur qui peut employer l'une des fonctions bitalloc , bitcalloc ou bitstrget . C'est au moment de l'allocation qu'est fixée la longueur maximale de la chaîne de bits, qui ne peut pas être changée.

La libération de la place allouée à un BITSTR se fait par la procédure bitfree .

Le paquetage bitstr définit aussi des procédures d'allocation et de libération de tableaux de chaînes de bits; un tel tableau est un pointeur vers un BITSTR et doit donc être déclaré par: BITSTR *t; La i-ème chaîne de t est désignée par t[i] .

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique