Rapport de Projet - Compilation

Sommaire

Introduction

Ce projet a été réalisé dans le cadre de notre formation universitaire de Licence, au cours du semestre 6 du parcours Informatique.

I. Objectifs

Les objectifs de ce projet sont au nombre de sept, parmis lesquels nous pouvons discerner cinq principaux de deux optionnels.

Les cinq objectifs principaux sont de réaliser :

1) un analyseur syntaxique du langage Pseudo-Pascal
2) un analyseur sémantique du langage Pseudo-Pascal
3) un interpréteur du langage Pseudo-Pascal
4) un traducteur (ou compilateur) de Pseudo-Pascal vers C3A
5) un interpréteur du langage C3A

Les deux objectifs optionnels sont de réaliser :

1) un traducteur (ou compilateur) de C3A vers Y86
2) un "ramasse-miettes" pour l'interpréteur du langage Pseudo-Pascal

Dans la suite, pour désigner le langage Pseudo-Pascal, nous ferons usage de l'acronyme "PP" pour des raisons de simplification.

II. Solutions mises en oeuvre

Pré-requis : quelques structures

Il est important d'indiquer que nous avons eu besoin d'implémenter certaines structures qui ont été nécessaires à la réalisation de plusieurs objectifs. Pour bien segmenter les différentes explications, nous parlerons de ces structures ici.

Dans la suite, si un fichier de sources (code C) est désigné, il se trouve nécessairement dans le dossier *"include"* à la racine du projet.

Les structures Variable et Type

Les déclarations de ces structures se trouvent dans le fichier "Variable.h" et leur implémentation dans "Variable.c".

Un Type est composé d'un entier *desc* faisant office de descripteur. Il se rattache lors de l'initialisation à l'une des constante suivante, désignant chacune un type :

```
#define BOOL 0
#define INT 1
#define ARRAY 2
#define VOID 3
```

Il dispose également d'un second membre *child* de type *struct Type** qui est son fils et qui permet d'utiliser la récursivité sur le type tableau.

Voici la déclaration de sa structure :

```
struct Type
{
   int desc;
   struct Type* child;
};
```

Une Variable est composée d'un type *type*, et de 3 valeurs entières :

- *value*, qui correspond à sa valeur ;
- *refs*, qui correspond à son adresse ;
- array_set, qui définit si la variable est une référence vers une variable existante (false) ou si elle est allouée en tant que nouvelle variable (true)

Voici la déclaration de cette structure :

```
struct Variable
{
   struct Type* type;
   int value;
   int refs;
   int array_set;
};
```

La structure environnement (Env)

La déclaration de cette structure se trouve dans le fichier *"Env.h"* et son implémentation dans *"Env.c".*

Un environnement est une table de hachage pour les variables de notre programme PP. Il se compose des éléments suivants :

- *names*, un tableau des noms des variables de l'environnement ;
- *keys*, un tableau de clés permettant d'accéder aux variables de façon cohérente et plus rapide qu'en utilisant des comparaisons de chaînes de caractères (équivalent d'un tableau d'adresses) ;

- *values*, un tableau de *Variables* (structure vue précédemment) ;
- *length*, la taille de l'environnement permettant de connaître les index des différents tableaux qui le composent.

Voici la déclaration de sa structure :

```
struct Env
{
   char** names;
   unsigned long* keys;
   struct Variable** values;
   int length;
};
```

La structure tas (Stack)

La déclaration de cette structure se trouve dans le fichier "Stack.h" et son implémentation dans "Stack.c".

Un stack est composé de 4 tableaux d'entiers :

- *values*, stocke les valeurs des variables. Dans le cas d'un tableau de valeurs, il y aura autant de valeurs que nécessaires stockées dans *values* ;
- *adr*, stocke l'index, dans *values*, des variables ajoutées au tas ;
- *size*, correspond, à la taille de chacune des variables ajoutées au tas (1 pour une variable de type booléenne ou entière, supérieur à 1 dans le cas d'un tableau) ;
- *refs*, est utile au *ramasse-miettes* (c.f. Objectif optionnel 2). Il contient pour chaque valeur ou groupe de valeurs présentes dans *values*, le nombre de variables qui la contiennent/référencent. Cette valeur est incrémentée à l'appelle de la méthode *Stack_ref()* et décrémentée à l'appelle de *Stack_deref()*.

Et de 2 entiers :

- *refsLength*, qui correspond à la taille des tableaux adr, size et *refs* ;
- *valuesLength*, qui, de la même façon, correspond à la taille du tableau *values*.

Voici la déclaration de sa structure :

```
struct Stack
{
   int* values;
   int* adr;
   int* size;
   int* refs;
   int refsLength;
   int valuesLength;
};
```

Exemple :
// Déclaration d'un tas vide
stack = Stack_init();
// État du stack
values = [ ]
adr = [ ]
size = [ ]
refs = [ ]
refsLength = 0
valuesLength = 0

// Lecture d'une entrée en PP
X = new array of Integer [4];
// État de stack
values = [**0,0,0,0**]
adr = [**0**]
size = [**4**]
refs = [**1**]
refsLength = **1**
valuesLength = **4**

// Lecture d'une nouvelle entrée en PP
Y = new array of Integer [2];
// État du stack
values = [0,0,0,0,**0,0**]
adr = [0,**4**]
size = [4,**2**]
refs = [1,**1**]
refsLength = **2**
valuesLength = **6**

A. Analyseur syntaxique du langage PP

Notre programme fait usage d'une structure "pilier" pour l'analyse syntaxique nommée arbre de syntaxe abstrait, que nous appellerons par la suite "AST" (Abstract Syntax Tree) pour plus de simplicité. La déclaration de cette structure se trouve dans *"Ast.h"* et son implémentation dans *"Ast.c".*

Un AST est composé :

- d'un entier *nodetype* qui détermine le type de noeud c'est-à-dire E pour expression, C pour commande, V pour variable, I pour constante... qui correspondent à des valeurs entières définies par une énumération que l'on peut trouver dans "*ppinterpret.tab.h*" lui-même issue de la compilation de "*ppascal.l*" ;
- d'un pointeur *value* qui correspondra à l'opération que le noeud représente (si *nodetype* vaut E, il pourra s'agir de "Pl" pour une addition par exemple) ou une

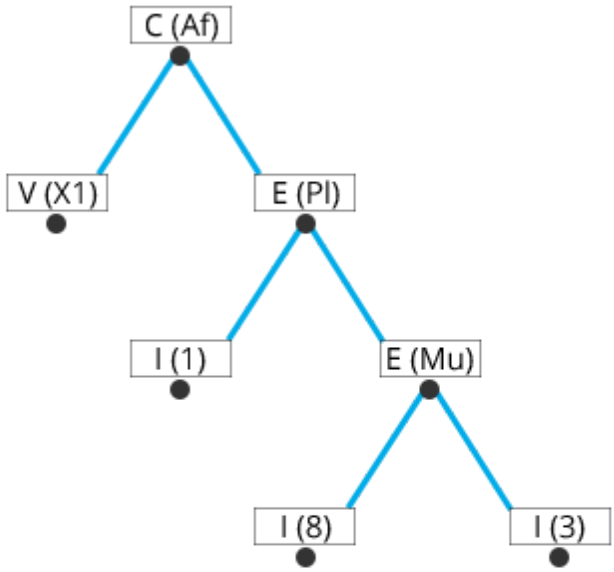
valeur typée (tableau, entier, booléen dans le cas où *nodetype* vaudrait I par exemple) ;
- d'un fils gauche et d'un fils droit, *left* et *right*, qui sont tous les 2 de type *struct Ast*.

Notons que dans le cas où notre noeud est en fait une feuille, c'est-à-dire que *nodetype* vaut **V** pour *variable* ou **I** pour *constante*, les fils *left* et *right* seront initialisés à 0.

Voici la déclaration de sa structure :

```
struct Ast
{
   int nodetype;
   void* value;
   struct Ast* left;
   struct Ast* right;
};
```

Dans le cas de la lecture d'une expression de ce type : **X1 := 1 Pl 8 Mu 3;**
Nous aurons un AST de cette forme :



La partie Bison de notre programme va, en fonction de la grammaire PP que nous lui avons spécifié, initialiser l'AST du code PP qui est lu en entrée.

Les différentes méthodes utilisées sont les suivantes :
struct Ast* Ast_init(int nodetype,int ope, struct Ast* left,struct Ast *right);
- Utilisée pour toute expression (E) et commande (C)

struct Ast* Ast_init_leaf_ptr(int nodetype, void* value);

- Utilisée lors d'une lecture de variable (V)

struct Ast* Ast_init_leaf_const(int nodetype, int value);

- Utilisée à la lecture d'une constante (I)

Exemple dans le cas E + E, extrait d'une analyse menée par bison :

```
E: E Pl E { $$ = Ast_init('E', Pl, $1, $3); }
```

Il s'agit ici d'initialiser un AST pour lequel :

- *nodetype* vaut 'E' ;
- *value* vaut 'Pl' ;
- *left* vaut '$1' c'est-à-dire 'E' qui sera à son tour évalué ;
- *right* vaut '$3' c'est-à-dire 'E' qui sera également traité (comme pour *left*).

Il peut s'agir d'une branche d'AST (fils d'un autre noeud).

Exemple dans le cas d'une définition de variable :

```
V { $$ = Ast_init_leaf_ptr('V', $1); }
```

Ici, une feuille d'AST est créée, c'est-à-dire que :

- *nodetype* vaut 'V' qui correspond au type "variable" ;
- *value* vaut '$1' c'est-à-dire sa valeur ;
- *left* et *right* valent tous les deux 0 puisqu'il s'agit d'une feuille de l'AST et non d'un noeud.

Ce code comporte cependant des fonctions et des valeurs que nous devons également gérer. C'est pourquoi il n'y a pas que des méthodes propres à la structure AST qui sont appelées dans la partie Bison, mais également des méthodes nécessaires à l'initialisation de ces fonctions et valeurs. Il s'agit des structures **Func** pour les fonctions, **FuncDisclaimer** lors d'une déclaration de fonction et **FuncList** qui est une liste de fonctions.

La déclaration de ces structures se trouve dans le fichier "Function.h" et son implémentation dans "Function.c".

Une déclaration de fonction est composée :

- d'un nom *name* de fonction ;
- d'un ensemble d'arguments *args*, une structure Env qui servira à constituer l'environnement local de la fonction. Il s'agit en fait des paramètres qui la compose ;
- d'un type de retour *type*.

En voici la structure :

```
struct FuncDisclaimer
{
        char* name;
        struct Env* args;
        struct Type* type;
```

```
};
```

Notons que le type dépend du *token* (symbole) lu par Bison tel que :
- *D_entf* qui correspondra à un type BOOL, INT ou ARRAY ;
- *D_entp* qui correspond au type VOID.

Une fonction est composée :
- d'un en-tête *disclaimer* (défini ci-dessus) ;
- d'un environnement *vars* qui correspond à l'environnement local de la fonction ;
- d'un AST *ast* qui contient les différentes instructions à exécuter par la fonction.

En voici la structure :

```
struct Func
{
   struct FuncDisclaimer* disclaimer;
   struct Env* vars;
   struct Ast* ast;
};
```

Une liste de fonctions est composée :
- d'un tableau de fonctions *list* ;
- d'une taille entière *length*.

En voici la structure :

```
struct FuncList
{
        struct Func** list;
        int length;
};
```

Exemple dans le cas d'une définition de procédure :
La démarche est ici différente puisqu'il ne s'agit pas simplement d'initialiser une arborescence d'AST mais bien une une fonction (ou procédure).

```
D: D_entp L_vart C { $$ = Func_init( $1, $2, $3); }
```

La méthode *Func_init* dont voici la macro :
struct Func* Func_init(struct FuncDisclaimer* disclaimer, struct Env* vars, struct Ast* ast), initialise une fonction à l'aide de son disclaimer ($1 = D_entp), d'une liste de variable ($2 = L_vart) et de son arbre syntaxique ($3 = C).

B. Analyseur sémantique du langage PP

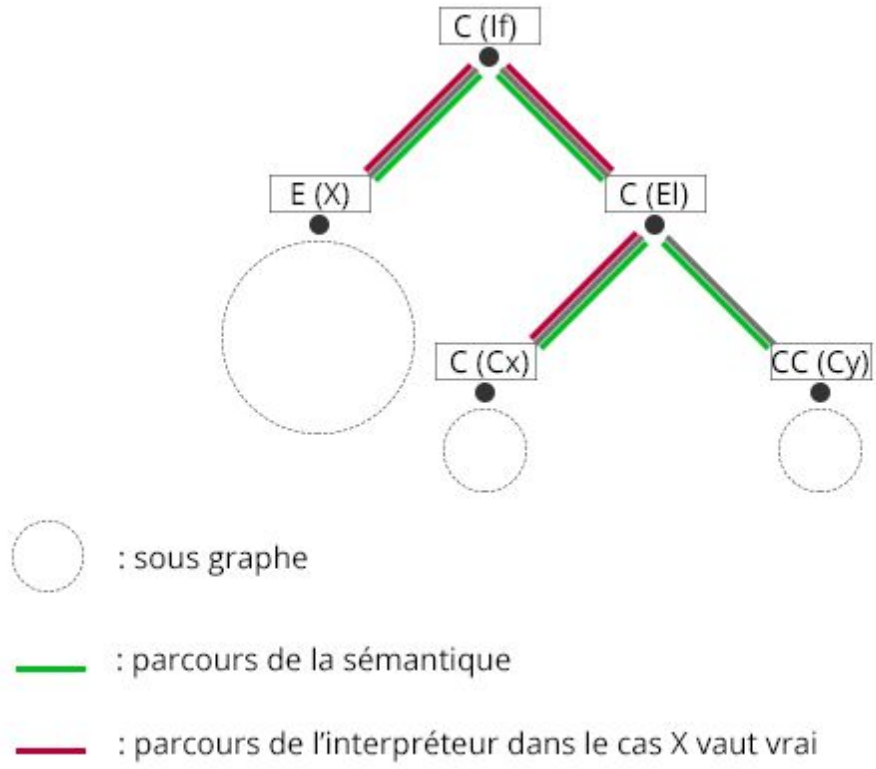
L'analyse sémantique de PP est effectuée par des fonctions dont la déclaration se trouve dans le fichier *"Pascal.h"* et l'implémentation dans *"Pascal.c".* Le nom de ces fonctions est préfixé par *"Pascal_semantic"*.

Le but de l'analyseur sémantique est de détecter des incohérences logiques du code lu en entrée. Il va donc parcourir l'ensemble des noeuds générés par l'analyse syntaxique réalisée au préalable et y effectuer les vérifications appropriées. Cela veut dire les vérifications associées au type d'opération principalement. Par exemple, vérifier la concordance des types, la non collision d'environnements (global et local), etc...

Cette analyse se fait en trois temps dans notre cas:

- Analyse des signatures des fonctions: Il s'agit ici de repérer les re-déclarations de fonctions, de vérifier leur cohérence et de ne conserver que la dernière déclaration. Ainsi, toutes les déclarations de fonction portant le même nom, doivent présenter la même signature (nombre et type des paramètres, type de retour). D'autre part, seule la dernière déclaration sera prise en compte.
- Analyse du corps du programme: On vérifie ici que le programme respecte des exigences sémantiques (l'addition ne s'applique que sur deux entiers, impossible de déclarer plusieurs fois la même variable dans les environnement globaux et locaux etc...)
- Analyse du corps des fonctions: Pour chaque fonctions, on analyse sa structure en vérifiant les mêmes contraintes que pour l'analyse du corps du programme.

A noter que l'ensemble des AST composant le programme est parcouru. C'est à dire que tous les embranchements possibles du programme (conditions, appels de fonction etc...) sont analysés.



C. Interpréteur du langage PP

L'interprétation de PP est effectuée par la fonction *"Pascal_run"* dont la déclaration se trouve dans le fichier *"Pascal.h"* et l'implémentation dans *"Pascal.c"*.

Notre interpréteur va, de façon semblable à l'analyseur sémantique, parcourir l'AST qui lui est passé. Il va effectuer sur l'environnement l'opération spécifiée par chaque noeud de l'AST qu'il doit parcourir. En effet, on ne parcourt pas forcément tous les noeuds, étant donné qu'à l'inverse de l'analyseur sémantique qui doit vérifier l'intégrité de l'entièreté du code, nous interprétons seulement la partie du programme qui doit l'être. Concrètement, ajouter des conditions à certaines parties du code PP a pour conséquence le fait que tout le code n'est pas forcément interprété.

L'environnement est donc modifié au fil de l'interprétation, avant d'être finalement affiché.

D. Compilateur PP vers C3A

La compilation de PP vers C3A est effectuée par les fonctions *"P_Compile_C3A"* et *"P_Compile_C3A_eval"* dont les déclarations se trouvent dans le fichier *"Pascal_C3A.h"* et les implémentations dans *"Pascal_C3A.c"*.

La première fonction prend en entrée un environnement, une liste de fonctions et un AST à partir desquels elle va lancer la compilation.

Dans un premier temps, les variables globales sont déclarées ainsi que des variables globales portant le nom des fonctions afin de gérer les valeurs de retour.

Dans un second temps, le corps du programme est compilé avec appels récursifs de P_Compile_C3A_eval sur l'AST.

Enfin, les différentes fonctions sont compilées à la suite du corps.

Le programme C3A résultant de cette opération de compilation est transmis, au fil de la compilation à la sortie standard.

E. Interpréteur du langage C3A

L'interprétation du langage C3A est effectuée par la fonction "*C3A_run*" dont la déclaration se trouve dans le fichier "C3A.h" et l'implémentation dans "C3A.c".

A l'inverse de l'interprétation du langage Pseudo-Pascal, l'interprétation du C3A se déroule en trois phases: Analyse lexicale, Analyse syntaxique et interprétation (on omet l'analyse sémantique qui n'est pas demandée). Nous ne détaillerons ici que les phases d'analyse syntaxique et d'interprétation. En effet, la phase d'analyse lexical est sensiblement similaire à celle de PPascal (dans sa logique), seuls diffèrent les lexèmes analysés et reconnus.

Analyse syntaxique

Cette fois-ci l'analyse syntaxique repose sur les structures Value, Quad et QuadList.

La structure Value, déclarée comme suit, représente une valeur.

```
struct Value
{
  char type;
  void *value;
};
```

Elle contient les champs:

- type: représente le type de la valeur tel que V (variable), I (constante entière) ou B (constante booléenne).
- value: Contient la valeur de la structure, dans le cas des constantes, ou le nom d'une variable de l'environnement dans le cas d'une variable.

Le Quad, dont la déclaration est visible ci-dessous, correspond dans les faits à une instruction C3A.

```
struct Quad
{
  char *address;
  int  operation;
  struct Value* arg1;
  struct Value* arg2;
  char* destination;
  struct Quad *next;
};
```

Un Quad présente les propriétés suivantes:

- address: Adresse de l'instruction dans le programme (en vue de l'appel vient un saut d'exécution)
- operation: Type d'opération à réaliser (Affectation, addition, saut, appel de fonction...)
- Les valeurs arg1 et arg2 qui correspondent aux arguments passés à l'instruction.
- destination: Qui indique soit le nom de la variable de l'environnement dans laquelle stocker le résultat de l'opération (dans la majeure partie des cas) ou l'adresse de la commande à atteindre dans le cas d'un saut (A noter que ce champs fait aussi office de troisième argument dans certains cas bien précis).
- next: Pointeur vers le prochain Quad à exécuter dans la suite logique du programme.

Enfin, la structure QuadList est assez simple, il s'agit en effet d'une liste de Quads:

```
struct QuadList
{
  struct Quad* start;
  struct Quad* end;
};
```

Le champs start présente le point d'entrée de la liste (son premier composant), tandis que le champs end correspond au dernier Quad de la liste.
Les propriétés de liste de la QuadList sont obtenus par effet de chaîne. Chaque Quad disposant d'un pointeur vers le prochain Quad à exécuter.

De par ses propriétés la QuadList représente un programme C3A complet, chacunes de ses entrées (Quad) correspondant à une instruction de celui-ci.

La partie Bison de notre programme va, en fonction de la grammaire C3A que nous lui avons spécifié, construire la QuadList représentant le programme C3A passé en entrée.

Par exemple le code Bison suivant va permettre l'initialisation d'un Quad (non-terminal O) réalisant une opération d'addition :

```
O : Pl Sp F Sp F Sp V { $$ = Quad_create(0,Pl, $3, $5, $7); }
```

De même, le code Bison présenté ci-dessous créer une QuadList à un élément (et alloue l'adresse V au Quad O).

```
C: V Sp O { $3->address = $1; $$ = QuadList_create($3);}
```

Cette QuadList a un élément sera concaténée récursivement à toutes les autres QuadList générées par Bison via le code:

```
C : C Se C { $$ = QuadList_concat($1, $3);}
```

Interprétation du code C3A

Le code C3A est exécuté par les fonctions C3A_run et C3A_eval décrites dans les fichiers "C3A.h" et "C3A.c". Fonctions auxquelles aux transmet, une liste de quadruplets (structure QuadList), son quadruplet de départ, un tas, ainsi que 3 environnements C3A qui sont l'environnement global, l'environnement local et l'environnement paramétrique.

L'interprétation du code C3A est relativement simple étant donné qu'il s'agit simplement de parcourir chacun des Quads présents dans la QuadList représentant le

programme, les uns après les autres, tout en réalisant les opérations auxquelles ils correspondent.

A noter cependant que cette exécution n'est pas parfaitement linéaire, en effet, certaines instructions (Jp, Jz) entraînent des sauts: on se rend directement aux Quads spécifiés dans ces instructions.

De même, les instructions Call et Ret présentent des sémantiques un peu particulières. En effet, Call entraîne un appel récursif sur la fonction ou procédure désignée de façon à ce qu'elle dispose d'environnements locaux et paramétriques appropriés. Il y a une copie de l'environnement paramétrique actuel dans le nouvel environnement local ainsi qu'une initialisation d'un nouvel environnement paramétrique qui seront affectés au prochain appel de "C3A_run".

III. Difficultés rencontrées

Analyse du sujet

La compilation représente pour nous en réalité une matière nouvelle et dont les composantes nécessaires sont également relativement nouvelles. En effet, nous n'apprenions qu'en parallèle certaines notions propres à la théorie des langages qui nous ont été nécessaires pour cette matière. La compilation constituant un sujet assez pointu de notre point de vue, et ne disposant que de peu de temps et d'un bagage relativement mince pour un tel projet, nous avons eu quelques soucis liés à la compréhension de ce dernier. Malgré une compréhension générale plutôt bonne, quelques nuances nous ont amené parfois à des pertes de temps considérables, ce qui était facteur de stress.

Une première implémentation du Stack inadaptée

La première implémentation de notre structure de tas n'était pas adaptée. Elle convenait lorsque nous étions sur du PP, mais le passage au C3A s'est trouvé être bloquant. En fin de compte, nous utilisions la structure Variable (qui a été modifiée également depuis cela) comme pouvant définir elle même un tableau dont les valeurs étaient poussées dans le tas. Chaque variable de type tableau disposait de sa taille et de son index de début dans le tas.

Les ramasses-miettes

Ayant à l'origine l'objectif de répondre aux consignes facultatives, nous avons implémenté bon nombre de logiques relatives à la gestion de la mémoire (ramasse-miette). Cependant, par manque de temps et suite à de nombreuses problématiques, ces logiques ne sont pas utilisées. Notre production se trouve donc être relativement gourmande en mémoire.

Remerciements

Nous souhaitons remercier notre professeur et chargé de TD, Monsieur Géraud SENIZERGUES, pour la qualité de l'enseignement qu'il nous a dispensé et de l'aide qu'il nous a fourni.

Ce projet a été réalisé par :

Benjamin de POURQUERY
Quentin LEMONNIER
Célia PAQUE
Clovis PORTRON