Projet de Compilation

CHARLET Guillaume GAUTIER Florian

10 avril 2014

Table des matières

1 Description du projet 2
2 Description des différents points à réaliser lors de ce projet . . . 3
2.1 Analyse syntaxique 3
2.2 Analyse sémantique 4
2.3 Interprétation du Pseudo-Pascal 4
2.4 Traduction en C3A 4
2.5 Interprétation du C3A 5
3 Mode d'emploi 6

1 Description du projet

Le projet de compilation consiste à analyser le langage Pseudo-Pascal, à l'interpréter, le traduire en C3A et à écrire un interprète de C3A.

Dans un premier temps nous devions faire l'analyse syntaxique du langage Pseudo-Pascal afin de récupérer :

— les variables globales
— la liste des fonctions et procédures avec leurs argurments, le type éventuel du résultat et leur corps
— le programme principal

Si le texte donné en entré n'est pas reconnu par la grammaire du Pseudo-Pascal, nous devions afficher un message d'erreur et arrêter la compilation du programme.

Ensuite, après l'analyse syntaxique nous devions faire l'analyse sémantique afin de vérifier que :

— toute expression est typable
— toute affectation a des membres de même type
— tout appel de fonction ou procédure a des paramètres d'appel qui ont la même suite de types que la suite des paramètres de la fonction ou procédure lors de sa déclaration

Si lors de l'analyse il existe des erreurs de sémantique, nous devions afficher un message d'erreur et ne pas continuer la compilation du programme puis, une fois les analyses terminées, nous devions interpréter le code Pseudo-Pascal, et afficher la valeur des variables globales une fois l'interprétation finie. Enfin, une fois cela fait, nous devions transformer le code Pseudo-Pascal en C3A, puis l'interpréter et afficher la valeur des variables globales.

Il nous était aussi possible de traduire le C3A vers le Y86, et un "ramasse-miettes" de l'interpréteur du code Pseudo-Pascal afin d'optimiser la mémoire pour la gestion des tableaux.

Pour réaliser ce projet nous devions être par groupes d'au plus 4 étudiants, et nous avions environ 3 semaines pour le faire. Enfin, nous avons choisi d'utiliser GIT comme gestionnaire de version afin de pouvoir travailler efficacement en groupe.

2 Description des différents points à réaliser lors de ce projet

2.1 Analyse syntaxique

Pour stocker le programme lors de l'analyse syntaxique nous avons créé nos propres structures et fonctions, que nous avons enregistrées dans les fichiers tree_abs.h et tree_abs.c.

Nous avons décidé de tout stocker dans un seul arbre, pour cela nous avons créé plusieurs structures :

Afin de définir le type d'une expression nous avons créé la structure "type_exp" contenant :

- la profondeur nommée "depth" qui contient le nombre d'éléments contenus dans le tableau suivant
- le tableau nommé "type" qui contient le type d'une expression, utilise une énumération "type_expression" {T_array, T_bool, T_int}

Par exemble :

- pour un entier "depth" vaut 1 et "type" vaut {T_int}
- pour un tableau de tableaux de booleens, "depth" vaut 3 et "type" vaut {T_array, T_array, T_bool}

Pour stocker l'arbre nous avons créé la structure "tree" contenant :

- une définition "def" qui permet de savoir à quoi vont correspondre ses fils et quelles opérations nous devrons faire dessus, le champ utilise une énumération "define" contenant la liste définitions possibles
- un type nommé "type" qui permet de définir le type de l'expression
- le nombre de fils nommé "nb_sons" qui permet de savoir combien il a de fils
- un tableaux de fils nommé "sons" qui contient les fils

Pour contenir la définition d'une variable nous avons créé la structure "var" contenant :

- un nom nommé "name" contenant le nom de la variable
- un type nommé "type" qui correspond au type de la variable

Pour contenir les valeurs constantes dans le programme nous avons créé la structure "val" contenant :

- une définition nommée "def" qui permet des savoir si c'est un nombre, une variable ou un boolean (true ou false) et qui utilise une énumération "type_value" {Bool, Int, Var}
- un paramètre nommé "param" qui contient sous la forme d'entier le nombre ou le boolean, ou sous la forme d'une chaine de caractères le nom de la variable

Ensuite nous avons créé plusieurs fonctions afin d'utiliser ces structure qui sont appellées lors de l'analyse syntaxique afin de récupérer toutes les informations.

2.2 Analyse sémantique

Pour l'analyse sémantique nous avons créé des fonctions dans tree_abs.c qui permettent de faire l'analyse du code.

Pour cela une fois l'analyse syntaxique finie il suffit d'appeller la fonction "analyze" avec en paramètre l'arbre généré par l'analyse syntaxique et qui retourne le nombre d'erreurs sémantiques dans le programme.

Si lors de l'analyse il y a une ou plusieurs erreur alors nous affichons de quel type d'erreur il s'agit et l'opération qui provoque cette erreur.

2.3 Interprétation du Pseudo-Pascal

Pour l'interpréteur du code en Pseudo-Pascal nous avons créé nos propres structures et fonctions, que nous avons enregistrées dans les fichiers interp.h et interp.c.

Nous avons alors défini une structure "env" pour contenir les variables, elle contient :

- le nom de la variable nommé "name"
- le type de la variable nommé "type"
- la valeur nommée "value"
- l'élément suivant nommé "next"

Et pour stocker les valeurs des tableaux nous avons créé la structure "heap" contenant :

- les adresses nommées "address" sous la forme d'un tableau d'entiers
- la taille des tableaux nommé "size" sous la forme d'un tableau d'entiers
- les cellules de chaque tableau nommé "memory" sous la forme d'une tableau d'entiers
- la première case disponible du tableau des adresses nommé "last_address"
- la première case disponible du tableau des cellule nommé "last_memory"
- le nombre d'erreurs d'accès mémoire nommé "error"

Puis nous avons créé plusieurs fonctions afin d'utiliser ces structures qui sont appellées lors de l'interprétation du code Pseudo-Pascal.

2.4 Traduction en C3A

Pour la traduction du code en Pseudo-Pascal en C3A nous avons créé nos propres structures et fonctions, que nous avons enregistrées dans les fichiers translate.h et translate.c.

Nous avons alors défini une structure "cell" qui correspond à une ligne de code C3A, contenant :
— l'étiquette nommée "name"
— l'opération C3A nommée "def" qui utilise une énumération "c3a" contenant la liste des opérations C3A
— l'argument 1 nommé "arg1"
— l'argument 2 nommé "arg2"
— la destination nommée "res"

Et une structure "list" qui contient la première et la dernière cellule C3A de la liste.

Puis nous avons créé plusieurs fonctions afin d'utiliser ces structures qui sont appellées lors de la traduction du code Pseudo-Pascal en C3A.

Pour la représentation des tableaux en C3A nous avons alors décidé d'utiliser un seul tableau d'entiers, donc lors de la traduction on ajoute une variable global "L_TAB#" qui contient l'indice de la première case libre. Cette variable sera modifiée lors de la traduction de la réservation de l'espace mémoire pour un tableau.

Nous avons aussi décidé que toute variable utilisée dans le code C3A n'étant pas dans l'environnement global est une variable locale de la partie du code évaluée.

Lors de l'appel d'une fonction le paramètre de destination d'un Call en C3A correspond à la valeur retournée par la fonction, alors que pour une procédure la valeur de la variable de destination prendra 0.

2.5 Interprétation du C3A

Pour l'interprétation du C3A nous avons créé nos propres structures et fonctions, que nous avons enregistrées dans les fichiers interp.h et interp.c.

Nous avons alors défini une struture "pile" qui correspond à une pile d'appel, contenant :
— la ligne du Call en C3A nommée "c"
— l'environnement local utilisé avant l'appel nommé "l"
— l'élément suivant dans la pile nommé "next"

Puis nous avons créé plusieurs fonctions afin d'utiliser les structures qui sont appellées lors de l'interprétation du C3A.

3 Mode d'emploi

La liste des fichiers nécessaires pour créer l'exécutable contient :
— ppascal.l (analyseur lexical)
— ppascal.l (analyseur syntaxique)
— tree_abs.h et tree_abs.c
— interp.h et interp.c
— translate.h et translate.c

Pour créer l'exécutable il suffit de taper la commande "make" dans le dossier contenant les fichiers nécessaires pour créer l'exécutable, cela va alors créer l'exécutable "ppascal".

Ensuite pour utiliser l'utiliser il suffit de lui donner sur l'entrée standard le code Pseudo-Pascal, par exemple "./ppascal < EXEMPLE/pex12.pp".

Lors de l'exécution on va alors afficher sur le terminal la liste des variables globales, la liste des fonctions et le code principal.

Ensuite il va afficher sur le terminal le résultat de l'analyse sémantique, c'est-à-dire soit que celle-ci a réussie, soit la liste des erreurs et provoquer l'arrêt du programme.

Enfin on va afficher l'environnement global de l'interprétation du code Pseudo-Pascal.

Pour la traduction du code en C3A le programme va alors créer un fichier "TRANSLATE_C3A.c3a" contenant la traduction du code, pour finalement afficher sur le terminal l'environnement global de l'interprétation du C3A.