Rapport Projet de Compilation

Alexis PICHON, Amélie RISI, Clément SIBILLE, Haize WEISS

10 avril 2017

Table des matières

1 Introduction

Le but de ce projet était de réaliser, en groupe, un analyseur syntaxique du langage Pseudo-Pascal, un analyseur sémantique, un interpréteur, un compilateur de Pseudo-Pascal vers C3A et enfin un interpréteur C3A.

Dans les parties qui suivent nous allons détailler ce qui a été fait et comment nous avons procédé, avant de traiter des différents problèmes rencontrés lors du projet.

2 Analyse syntaxique

Pour l'analyse syntaxique, nous avons repris le travail réalisé lors du TD concernant les tableaux.

2.1 La grammaire

Nous avons ensuite modifié le lexer et le fichier Bison pour obtenir la grammaire voulue. Nos conventions d'écriture sont les suivantes, les non-terminaux sont écrits en lettres minuscules et les terminaux exclusivement en lettres majsucules. La grammaire, fournie initialement dans le sujet du projet, était ambiguë. Pour pallier à ce problème, le non-terminal E de la grammaire initiale a été transformé en expr, term et fact, des non-terminaux qui qualifient respectivement l'expression, le terme et le facteur.

2.1.1 L'expression

L'expression est le dernier échellon de priorité des opérateurs, elle englobe donc tous les opérateurs qui ont une faible priorité : Plus, Moins, Ou, Inférieur strictement, Egal. Ce sont des terminaux écrits respectivement de cette manière dans notre grammaire : PL, MO, OR, LT, EQ.

2.1.2 Le terme

Le terme répresente une plus forte priorité par rapport aux précédents. Il correspond aux opérateurs suivants : Multiplication, Et, Non. Ces opérateurs sont écrits de telles manières dans notre parser : MU, AND, NOT.

2.1.3 Le facteur

Le facteur représente une expression parenthésée, un entier, un indentificateur de variable, un booléen, une déclaration de fonction, une déclaration de tableaux. Les terminaux présent sont donc I pour l'entier, V pour un indentificateur de variable, B pour un booléen, NEWAR pour la déclaration d'un tableaux.

2.1.4 Les commandes

Afin de lever une ambiguïté due à la séquence SE, nous avons spécifié qu'une commande engendre une séquence d'une commande suivie d'une commande atomique. Une commande atomique est soit une affectation AF, soit un skip SK, soit un if..then..else, IFTHEL avec une expression pour une commande ou une commande atomique, soit un while..do, WHDO avec une expression pour une commande atomique.

Voici donc notre grammaire respectant celle du projet mais avec des noms de non-terminaux un peu plus explicites.

```
prog:   block_decl_typed_var list_def cmd

expr:   expr PL term
        | expr MO term
        | expr OR term
        | expr LT term
        | expr EQ term
        | term

term:      term MU  fact
        | term AND fact
        | NOT fact
        | fact
fact:      '(' expr ')'
        | I
        | V
        | B
        | V '(' block_expr ')'
        | NEWAR type_decl '[' expr ']'
        | typed_expr
```

```
typed_expr: V  '[' expr ']'
            | typed_expr '[' expr ']'

cmd:        cmd SE atomic_cmd
            | atomic_cmd
atomic_cmd: typed_expr AF expr
            | V AF expr
            | SK
            | '{' cmd '}'
            | IF expr TH cmd  EL atomic_cmd
            | WH expr DO atomic_cmd
            | V '(' block_expr ')'

block_expr: %empty
            | block_non_nil_typed_expr

block_non_nil_typed_expr:   expr
                          | expr ',' block_non_nil_typed_expr

typed_arg:      V ':' type_decl

/* un type */
type_decl:      T_BOO
                | T_INT
                | T_AR type_decl

block_decl_typed_var:   %empty
                      | block_decl_non_nil_typed_var

block_decl_non_nil_typed_var:   VAR typed_arg
                              | block_decl_non_nil_typed_var ',' VAR typed_arg

block_def_proc:     DEP IDPROC '(' block_expr ')'
block_def_func:     DEF IDFUNC '(' block_expr ')' ':' type_decl

decl_def:   block_def_proc block_decl_typed_var cmd
            | block_def_func block_decl_typed_var cmd

list_def:   %empty
            | list_def decl_def
```

Par conséquent, nous allons vous détailler la liste des non-terminaux.

- prog : programme principal
- block_decl_typed_var : bloc de déclaration de variables typées
- list_def : liste de définition de fonctions ou procédures
- cmd : une commande
- expr : une expression sur un entier, un booléen ou un tableau
- term : terme d'une expression
- fact : facteur d'une expression
- block_expr : bloc d'expression
- type_decl : type déclaré
- typed_expr : expression typée
- atomic_cmd : commande atomique (commande non décomposable)
- block_non_nil_typed_expr : bloc de variables typées non nulles
- typed_arg : argument typé
- block_decl_non_nil_typed_var : bloc de déclaration de variables typées non nulles
- block_def_proc : bloc de définition de procédures
- block_def_func : bloc de définition de fonctions
- decl_def : définition d'une fonction ou procédure

3 Analyse sémantique

Nous avons repris les TP sur l'analyse sémantique et la gestion des tableaux.

Donc, nous utilisons les modules Arbre et Anasem.

3.1 Module Arbre

Ce module permet de gérer l'environnement du programme par le biais d'une biliste. On peut donc créer des noeuds pour construire un arbre syntaxique abstrait afin de renvoyer celui-ci lorsque l'analyse sémantique est terminée.

3.2 Module Anasem

Le module Anasem, nous permet de gérer les types. Nous renvoyons une erreur lorsque l'on constate une incohérence de typage.

4 Interpréteur PP

L'interpréteur de Pseudo Pascal va recevoir un arbre de syntaxe abstraite ainsi que l'environnement qui lui est associé et en produire une interprétation. Cette tâche est réalisée par le module interp.

4.1 interp

Le module interp reprend les concepts de base fournis à l'occasion du TD06 et en constitue une version permettant d'interpréter correctement du pseudo pascal (hors fonctions/procédures). Pour ce faire, une analyse descendante et en profondeur de l'arbre syntaxique est effectuée.

4.2 évaluation

Afin d'interpréter correctement le pseudo pascal, il est necéssaire d'évaluer chaque noeud. En effet, certains noeuds comme ceux impliquant des calculs peuvent subir une évaluation en vue d'être réduits. Cela signifie qu'après évaluation tout noeud dont la réduction aura été possible pourra être affiché. De ce fait, l'interprétation effectuera tout calcul possible afin de décrire l'état de la mémoire et des variables au moment de la fin du programme.

5 Compilateur PP vers C3A

Pour traduire un programme écrit en langage PP vers le langage C3A, nous sommes partis sur la base de la correction du compilateur IMP vers C3A réalisé lors du mini-projet. Afin de produire du code C3A, il est nécessaire de reprendre la structure en arbre utilisée au cours des étapes précédentes en vue de la revoir. Le code à 3 adresses est caractérisé par une liste de quadruplets qui illustrent le modèle vu en cours : Etiquette : Opérateur : Argument1 : Argument2 : Résultat.

Chaque opération recevra ou non des arguments enregistrés dans le quadruplet (un argument pouvant correspondre à l'étiquette d'un autre quadruplet, notamment pour l'initialisation d'une variable). Elle pourra également transmettre ou non un résultat pouvant ensuite être utilisé par d'autres opérations ou référencé au sein d'autres quadruplets. Afin de permettre une clarification du produit de l'analyse syntaxique, le code à 3 adresse va également référencer toute étape implicite requise par le programme. Ainsi, lorsqu'une variable sera initialisée, du code à 3 adresses illustrant le chargement de la valeur dans un registre en vue d'une affectation sera produit. Cela signifie qu'un noeud correspondant à une expression terminale (telle que l'affectation d'un numéral à une variable) équivaudra à 2 instructions en C3A.

6 Interpréteur C3A

Après traduction d'un programme PP en code à 3 adresses, l'interpréteur sera en mesure d'afficher la biliste de quadruplets correspondant au programme analysé et traduit. Cette interprétation prenant en compte les étapes implicites que le code d'origine ne déclare pas, l'interpréteur recevra une liste au sein de laquelle ces instructions supplémentaires seront intégrées avant l'instruction traduite du PP. Le référencement des variables, constantes et états de l'automate repose sur un compteur similaire à un compteur ordinal. Celui-ci permettra de retracer la pile des appels engendrés par une instruction. Par exemple, une affectation d'un numéral dans une variable sera représentée par des instructions dont la dernière (l'affectation finale) aura une valeur de compteur la plus faible.

7 Problèmes rencontrés

Le principal problème rencontré est la gestion des fonctions et des procédures. Etant donné que nous avons pas réussi à créer un parseur fonctionnel avec celles-ci, et par manque de temps, nous avons, par conséquent, décidé de travailler sur un « pseudo-pseudo-pascal » qui ne prend pas en compte les fonctions et procédures. Néanmoins, en dehors de ce point problématique, nous avons respecté la définition du langage pseudo-pascal du projet.

De plus, nous avons eu du mal à régler un problème lors de l'analyse sémantique sur un tableaux. En effet, lors de l'affectation, nous avions un décalage dans le TAS. Ce problème a été régleé par simple affectation à la bonne addresse.

Enfin, nous avons encore un problème persistant lors de l'analyse semantique lorsqu'il faut gérer le typage.

8 Conclusion

Ce projet de compilation, nous a permis d'intégrer toutes les spécificités vues en cours concernant la réalisation d'un compilateur et d'un interpréteur pour un langage donné. De plus, il nous a permis d'acquérir plus de notion sur comment effectué une analyse syntaxique avec des types, ainsi que d'utiliser le mode à grand pas pour l'analyse sémantique.