Rapport de projet de compilation

Encadrant : M. Da Silva Sébastien

Gary Guyot

Alexandre Farnier

Nicolas Bédrine

Vincent Albert

14 Mai 2015

Table des matières

Chapitre 1

Introduction

Pour rappel, l'objectif de ce projet était d'écrire un coompilateur pour le langage Tiger, décrit par Andrew Appel, fonctionnant sur l'émulateur de machine RISK créé par Alexandre Parodi, microPIUP. Pour ce faire, nous avons eu recours à ANTLR pour la spécification de la grammaire et la génération de l'arbre syntaxique (AST), au langage Java pour faire l'analyse sémantique, ainsi que pour générer le code Assembleur au format microPIUP/ASM. Pour la gestion des versions et le travail en équipe, nous avons utilisé git et déposé tous nos fichiers sur la forge de TELECOM Nancy.

Ce projet a été réalisé en quatuor, composé de Vincent Albert, Alexandre Farnier, Nicolas Bédrine et Gary Guyot, a commencé en Janvier 2016, et a été encadré par M. Da Silva Sébastien.

Chapitre 2

Spécification grammaticale du langage Tiger

2.1 La grammaire sous ANTLR

Pour spécifier le langage à l'aide d'ANTLR, nous avons dû travailler sur un seul fichier ; c'est pourquoi nous avons choisi de le réaliser tous ensemble, à chaque fois sur un seul et même ordinateur. L'objectif était de rendre la grammaire LL(1). Nous avons tout d'abord tenté de transcrire directement la grammaire telle qu'elle était spécifiée dans le sujet, puis de la retravailler directement avec ANTLRworks. Ceci s'est soldé par un échec. Nous avons ensuite tenté de transcrire les règles les plus fondamentales de la spécification fournie, puis d'implémenter au fur et à mesure les règles en la rendant LL(1). Cependant, nous n'avions pas pensé aux priorités d'opérations pour la génération de l'AST à ce moment là et nous nous sommes retrouvés face à une impasse. Notre troisième solution a été de partir de notre propre spécification des expressions arithmétiques simples puis d'empiler petit à petit les règles grammaticales de Tiger. Cette stratégie s'est avérée payante, et a conduit au fichier Tiger3.g dont voici le contenu :

```
grammar Tiger3;

options{
k=1;
backtrack=false;
output=AST;

}

tokens{
TAIGA;
DECLARATIONS;
BLOCK;
COMP;
COND;
BEGIN;
END;
PARAMSFORM;
PARAMSEFF;
PARAM;
```

```
TYPE;
TAB;
FIELD;
STRUCT;
CELL;
SIZE;
INIT;
PRIMITIF;
IDF;
CONST;
FUNC_DECL;
FUNC_CALL;
EMPTY_SEQ;
NEG;
}

@header {
import java.util.HashMap;
}

@members {
/** Map variable name to Integer object holding value */
HashMap<String,Integer>  memory = new HashMap<String,Integer>();
}

tiger3: e1=expr -> ^(TAIGA $e1);

expr:    nilexp
|        affect
|        type_id ( '{' field_list* '}')?
|        ifop
|        forop
|        whileop
|        breakexp
|        l=letexp  decl=declaration_list?  inexp e=expr_seq?  endexp
-> ^($l ^(DECLARATIONS $decl)? ^(BLOCK $e)?)
;

expr_list:       e1=expr (',' e2=expr_list)?
-> $e1 $e2?
;

expr_seq:        e1=expr  (pv=';'   e2=expr_seq)?
-> $e1 $e2?
;

field_list:      i1=ID ':=' e1=expr (v=',' f=field_list)?
-> {$v.text != null}? ^($i1 $e1) $f
->  ^($i1 $e1)
;

ifop:    fi=ifexp e1=expr   th=thenexp   e2=expr   (els=elseexp   e3=expr)?
-> ^($fi ^(COND $e1) ^($th $e2) ^($els $e3)?)

;

forop:  fo=forexp dd=ID ':=' e1=expr toexp e2=expr   doexp   e3=expr
-> ^($fo ^($dd ^(BEGIN $e1) ^(END $e2)) ^(BLOCK $e3))
;

whileop:         whi=whileexp e1=expr   doexp   e2=expr
-> ^($whi ^(COND $e1) ^(BLOCK $e2))
;

affect: o=orop (af=':=' e1=expr)?
-> {$af != null}? ^($af $o $e1)
-> $o
;

orop:    a1=andop (ortoken='|'^ andop)*
;

andop:   c1=comp (andtoken='&'^ comp)*
;

comp:
b1=binary ((sup1='>'(eg1='=')? | inf1='<' (sup2='>' | eg2='=')? | eg3='=') b2=binary)?
-> {$sup1 != null && $eg1 != null}? ^(COMP[">="] $b1 ^($b2))
```

```
-> {$inf1 != null && $eg2 != null}? ^(COMP["<="] $b1 ^($b2))
-> {$inf1 != null && $sup2 != null}? ^(COMP["<>"] $b1 ^($b2))
-> {$sup1 != null }? ^($sup1 $b1 ^($b2))
-> {$inf1 != null }? ^($inf1 $b1 ^($b2))
-> {$eg3 != null}? ^($eg3 $b1 ^($b2))
-> $b1
;
binary: b2=binary2 (addminexp^ b21=binary2)*
;

binary2:        n1=neg ((mul='*'^|div='/'^) neg)*
;

neg:    minus='-'? a=atom
-> {$minus != null}? ^(NEG $a)
-> $a
;

atom:   '(' e=expr_seq? ')'
-> {$e.tree != null}? $e
-> EMPTY_SEQ
|       lvalue
|       INT
|       STRING
;

lvalue:
i=ID (v=lvalue2 | par='(' e=expr_list? ')' | acc='{' l=field_list? '}')?
-> {$par.text != null && $e.tree != null}? ^(FUNC_CALL $i ^(PARAMSEFF $e))
-> {$par.text != null}? ^(FUNC_CALL $i) //Appel de fonction sans params
-> {$v.tree != null}? ^($i $v)  //AccÃ¨s tableau ou champ de structure
-> {$acc.text != null}? ^($i $l) //Instanciation d'une structure
-> $i
;

lvalue2:        '.' ID v=lvalue2? -> ^(FIELD ID $v?)
|       '[' e1=expr ']' (val=lvalue2 | o=ofexp e2=expr)?
->  {$o.text != null}? ^(SIZE $e1) ^(INIT $e2)  //Initialisation de tableau
-> {$val.tree != null}? ^(CELL $e1) $val //Successio de lval
-> ^(CELL $e1) //AccÃ¨s tableau
;

declaration_list:       d1=declaration (d2=declaration_list)?
-> $d1 $d2?
;

declaration:    type_declaration
|       variable_declaration
|       function_declaration
;

type_declaration
:       t1=typeexp i=ID '=' t2=type
-> ^($t1 $i $t2)
;

type:   (ID | t=type_id)
-> {$t.tree != null}? ^(PRIMITIF $t)
-> ^(PRIMITIF ID)
|       '{' t=type_fields? '}'
-> {$t.tree != null}? ^(STRUCT $t)
-> ^(STRUCT)
|       'array of' (t=type_id | i=ID)
-> {$t.text != null}? ^(TAB $t)
-> ^(TAB $i)
;

variable_declaration
:       vava=varexp nom=ID ( depoi=':' (typenew=ID | typebase=type_id))? ':=' e=expr
-> {$depoi != null && $typenew.text!=null}? ^($vava $nom $typenew $e)
-> {$depoi != null && $typebase.text!=null}? ^($vava $nom $typebase $e)
-> ^($vava $nom $e)
;

function_declaration
:       functionexp ID '(' par=type_fields? ')' (':' (ty=type_id|i=ID))?  '='   (e=expr  )+
-> {$par.text != null && $ty.text != null}? ^(FUNC_DECL ID ^(PARAMSFORM $par) ^(TYPE $ty) ^(BLOCK $e))
-> {$par.text != null && $i != null}? ^(FUNC_DECL ID ^(PARAMSFORM $par) ^(TYPE $i) ^(BLOCK $e))
```

```
-> {$par.text != null}? ^(FUNC_DECL ID ^(PARAMSFORM $par) ^(BLOCK $e))
-> {$ty.text != null}? ^(FUNC_DECL ID ^(TYPE $ty) ^(BLOCK $e))
-> {$i != null}? ^(FUNC_DECL ID ^(TYPE $i) ^(BLOCK $e))
-> ^(FUNC_DECL ID ^(BLOCK $e))
;

type_fields:     t1=type_field t2=type_fields2?
-> $t1 $t2?
;

type_fields2:    ',' t1=type_field t2=type_fields2?
-> $t1 $t2?
;

type_field:      i1=ID ':' (t=type_id|i2=ID)
-> {$i2.text != null}? ^(PARAM["p"] $i1 $i2)
-> ^(PARAM["p"]$i1 $t)
;

type_id:         (i='int' | s='string') lvalue2*
;

arrayexp:        'array' ;
breakexp:        'break' ;
doexp:   'do'            ;
elseexp:         'else'          ;
endexp:  'end'           ;
forexp:  'for'           ;
functionexp:     'function'      ;
ifexp:   'if'            ;
inexp:   'in'            ;
nilexp:  'nil'           ;
letexp:  'let'           ;
ofexp:   'of'            ;
thenexp:         'then'  ;
toexp:   'to'            ;
typeexp:         'type'  ;
varexp:  'var'           ;
whileexp:        'while' ;
typedefexp:      'typedef' ;
blockexp:        'block' ;
addminexp:       '+' | '-';

ID       :       ('a'..'z'|'A'..'Z')(('a'..'z'|'A'..'Z'|'0'..'9'|'_')*);
INT      :       '0'..'9'+;
STRING   :       '"'.+'"'; /* . correspond Ãă n'importe quel caractÃÍre ou n'importe quel caractÃÍre affichable ?*/
WS       :       (' '|'\t')+ {$channel=HIDDEN;};
NEWLINE  :       '\r'? '\n' {$channel=HIDDEN;};
COMMENT  :       '/*'.* '*/' {$channel=HIDDEN;};
COMMENT2         : '//' .* '\r'? '\n' {$channel=HIDDEN;};
```

2.2 L'arbre syntaxique

Chapitre 3

L'analyse sémantique

3.1 La structure de la table des symboles

3.2 Les erreurs sémantiques

Chapitre 4

La génération de code assembleur

4.1 Quelques schémas de traduction

4.2 Exemples d'application

Chapitre 5

Répartition du travail

Chapitre 6

Conclusion