INFO-F403 Introduction to Language Theory and Compilation

Chapeaux Thomas
Dagnely Pierre

March 14, 2013

Le but du projet était de construire un compilateur d'une version simplifiée de Perl en ASM/ARM devant tourner dans une architecture Android.

La première partie de ce rapport se concentrera sur l'analyse du language, c'est-à-dire la définition des tokens et de la grammaire LL(1) correspondante, ainsi que la table d'action et les contraintes que nous avons imposé lors de la transformation en LL(1).

Ensuite, nous décrirons notre parser qui transforme un fichier .perl en un AST.

Finalement, nous expliquerons la génération du code ASM/ARM.

1 Language

1.1 Lexèmes

Définition des tokens

Lexical units	regular expressions
INT	([0-9])*
FLOAT	([0-9])*.DOT.([0-9])*
BOOL	(0+1+true+false+")
STRING	'.([A-Za-z]+[0-9])*.'
FAC	!
MUL	*
DIV	/
MINUS	-
ADD	+
LT	<
GT	>
LE	<=
GE	>=
EQUIV	==
DIF	!=
AND	&&
OR	||
NOT	not
LT-S	lt
GT-S	gt
LE-S	le
GE-S	ge
EQ-S	eq
NE-S	ne

Lexical units	regular expressions
EQUAL	=
DOT	.
SEMICOLON	;
COMA	,
OPEN-PAR	(
CLOSE-PAR	)
OPEN-BRAC	{
CLOSE-BRAC	}
OPEN-COND	IF
CLOSE-COND	ELSE
ADD-COND	ELSIF
NEG-COND	UNLESS
RET	return
FUNCT-DEF	SUB
ID	STRING
FUNCT-NAME	&.STRING
PERL-DEF	defined
PERL-INT	int
PERL-LENG	length
PERL-SCAL	scalar
PERL-SUBS	substr
PERL-PRIN	print
COMM	#.STRING
VARIABLE	$.STRING

Remarques :

- Le lexème COMA peut définir l'opérateur virgule ou juste une virgule entre deux paramètres. Le lexème est le même mais le parser l'interpretera différemment.

- Les lexèmes de type COMM (commentaire) sont supprimés avant de faire l'analyse lexicale

1.2 Automates

Définition des automates finis

DFA

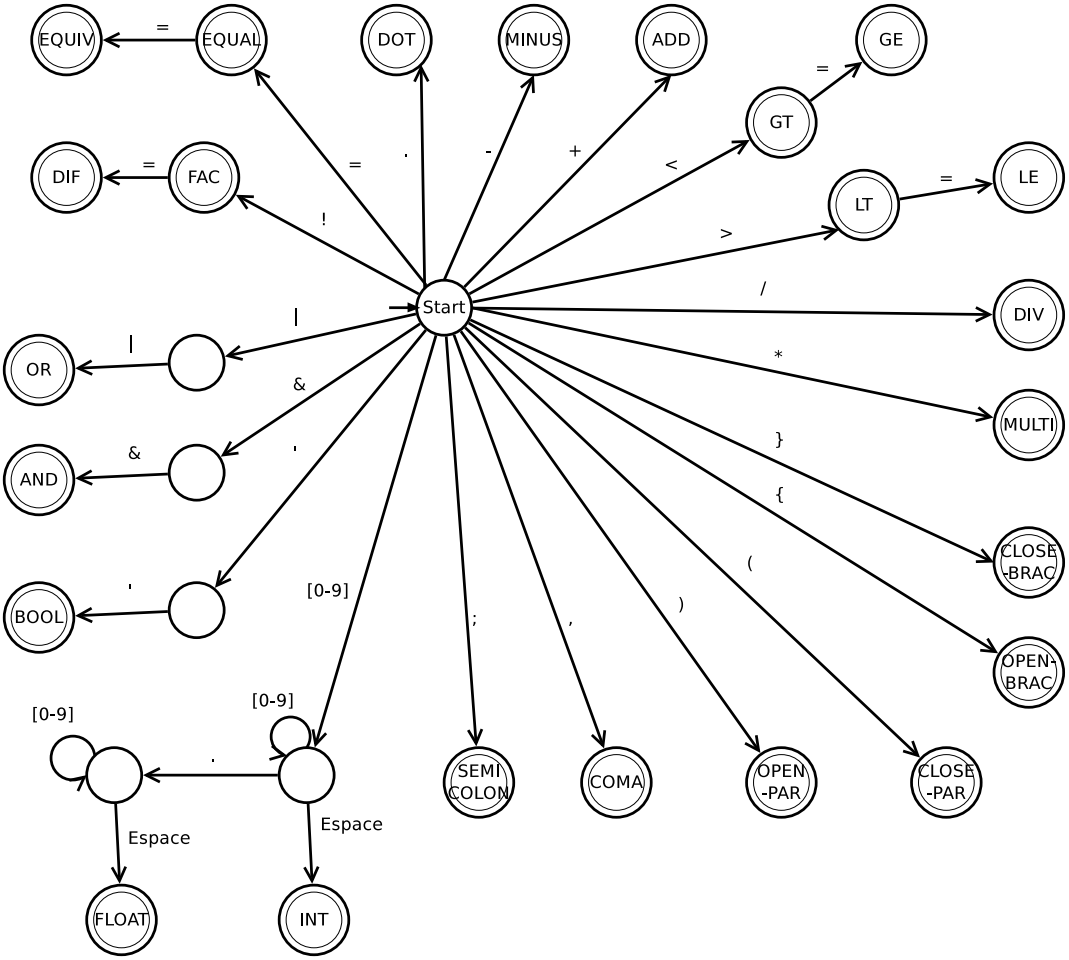

Figure 1: automate "non alphabétique"

Dans la figure 2, Les flèches bleues représentent les transitions vers l'état ID.

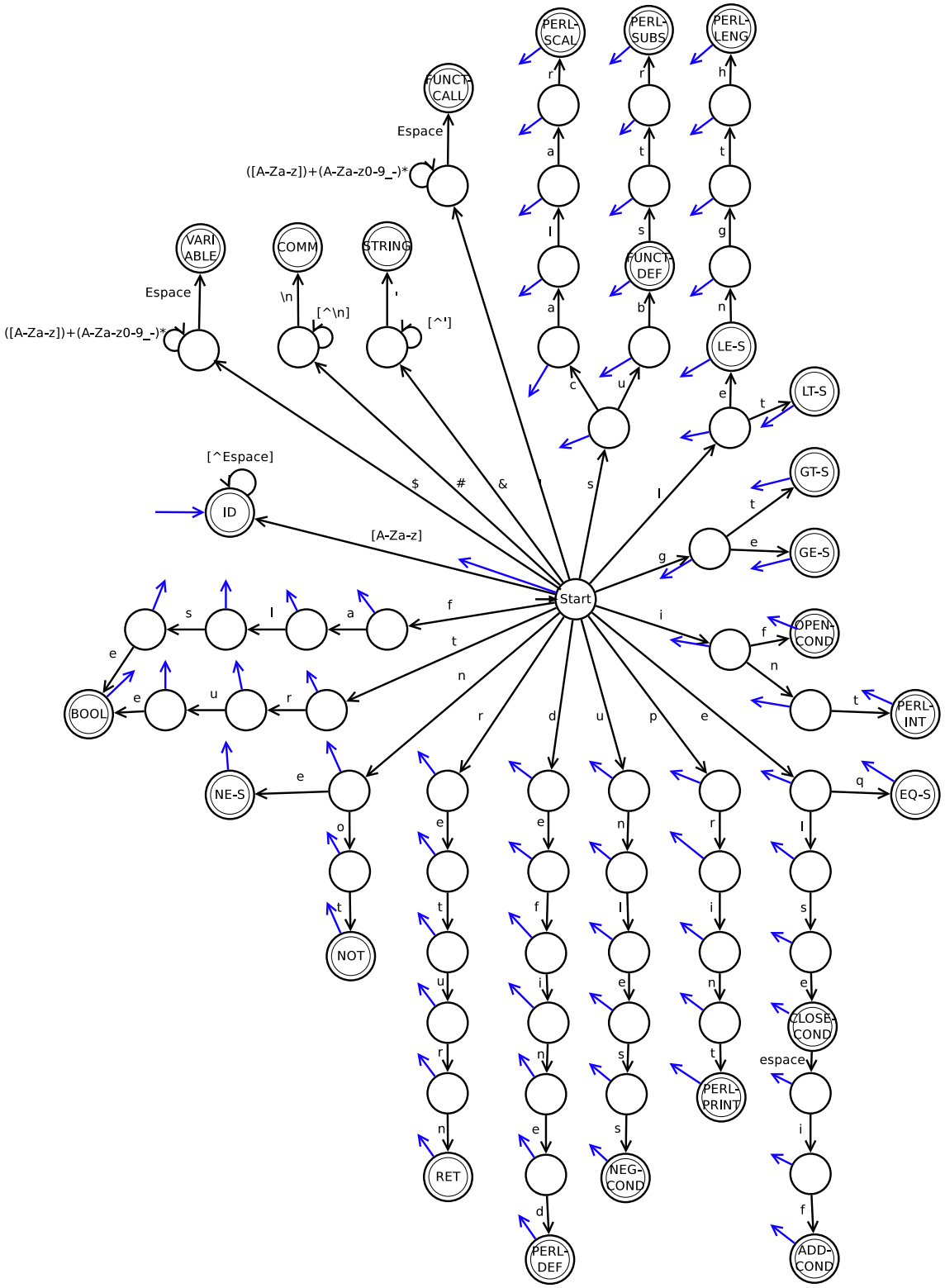

Figure 2: automate "alphabétique"

1.3 Grammaire

1.3.1 Grammaire initiale

Pour rappel, voici la grammaire donnée initialement :

<PROGRAM> → <FUNCT-LIST> <INSTRUCT-LIST>
→ <FUNCT-LIST>
→ <INSTRUCT-LIST>
<FUNCT-LIST> → <FUNCT>
→ <FUNCT-LIST> <FUNCT>
<FUNCT> → funct-def id open-par <FUNCT-ARG> close-par open-brac <INSTRUCT-LIST> close-brac
<FUNCT-ARG> → open-par <ARG-LIST> close-par
<ARG-LIST> → <ARG-LIST> coma variable
→ variable
→ epsilon
<INSTRUCT-LIST> → <INSTRUCT-LIST> <INSTRUCT> semicolon
→ <INSTRUCT> semicolon
<FUNCT-CALL> → funct-name <FUNCT-CALL-ARG>
<FUNCT-CALL-ARG> → <FUNCT-CALL-ARG> coma <EXP>
→ <EXP>
→ epsilon
<INSTRUCT> → variable equal <EXP>
→ <EXP>
→ ret <EXP>
→ <COND>
<COND> → open-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
<COND-END> → add-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
→ close-cond open-brac <INSTRUCT-LIST> close-brac
→ epsilon
<SIMPLE-EXP> → int
→ <FUNCT-CALL>
→ variable
→ string
<EXP> → <SIMPLE-EXP>
→ open-par <EXP> close-par
→ <EXP> add <EXP>
→ <EXP> minus <EXP>
→ <EXP> multi <EXP>
→ <EXP> div <EXP>
→ <EXP> equiv <EXP>
→ <EXP> gt <EXP>

1.3.2 Suppression des symboles inutiles

On peut retirer les symboles non-productifs et les symboles inaccessibles. Ici tous les symboles sont utiles dont on ne peut en retirer aucun.

1.4 Gestion des priorités et associativité

On veut retirer les ambiguités liées aux priorités et à l'associativité. Cela ne concerne que la règle EXP, on la transforme donc en respectant les règles de priorités et d'associativités habituelles :

<EXP> → <EXP> equiv <EXP-2>
→ <EXP> gt <EXP-2>
→ <EXP-2>
<EXP-2> → <EXP-2> add <EXP-3>
→ <EXP-2> minus <EXP-3>
→ <EXP-3>
<EXP-3> → <EXP-3> mul <SIMPLE-EXP>
→ <EXP-3> div <SIMPLE-EXP>
→ <SIMPLE-EXP>

1.5 left factoring

Cela ne concerne que EXP, EXP-2, EXP-3 et PROGRAM :

<PROGRAM> → <FUNCT-LIST> <PROG-TAIL>
→ <INSTRUCT-LIST>
<PROG-TAIL> → <INSTRUCT-LIST>
→ epsilon

<EXP> → <EXP-2> <EXP-TAIL>
<EXP-TAIL> → equiv <EXP-2> <EXP-TAIL>
→ gt <EXP-2> <EXP-TAIL>
→ epsilon
<EXP-2> → <EXP-3> <EXP-2-TAIL>
<EXP-2-TAIL> → add <EXP-3> <EXP-2-TAIL>
→ minus <EXP-3> <EXP-2-TAIL>
→ epsilon
<EXP-3> → <SIMPLE-EXP> <EXP-3-TAIL>
<EXP-3-TAIL> → mul <SIMPLE-EXP> <EXP-3-TAIL>
→ div <SIMPLE-EXP> <EXP-3-TAIL>
→ epsilon

1.6 Left recursion

Les grammaires LL(k) ne peuvent contenir de règles du type $A \rightarrow A\beta$. Cela concerne FUNCT-LIST, ARG-LIST, INSTRUCT-LIST, FUNCT-CALL-ARG :

<FUNCT-LIST> → <FUNCT-LIST-BEG> <FUNCT-LIST-END>
<FUNCT-LIST-BEG> → <FUNCT>
<FUNCT-LIST-END> → <FUNCT> <FUNCT-LIST-END>
→ EPSILON

<ARG-LIST> → <ARG-LIST-BEG> <ARG-LIST-END>
<ARG-LIST-BEG> → variable
→ epsilon
<ARG-LIST-END> → coma variable <ARG-LIST-END>
→ epsilon

<INSTRUCT-LIST> → <INSTRUCT> semicolon <INSTRUCT-LIST>
→ epsilon

<FUNCT-CALL-ARG> → <FUNCT-CALL-ARG-BEG> <FUNCT-CALL-ARG-END>
<FUNCT-CALL-ARG-BEG> → <EXP>
→ epsilon
<FUNCT-CALL-ARG-END> → coma <EXP> <FUNCT-CALL-ARG-END>
→ epsilon

1.7 Suppresion des productions unitaires

règle FUNCT-ARG (<FUNCT-ARG> → open-par <ARG-LIST> close-par) est dans ce cas
On la remplace donc directement par ARG-LIST dans funct.

FUNCT devient donc : (<FUNCT> → funct-def id <ARG-LIST> open-brac <INSTRUCT-LIST> close-brac)
et arg-list devient : (<ARG-LIST> → open-ar <ARG-LIST-BEG> <ARG-LIST-END> close-par)
supprime donc une règle pas intermédiaire peu utile

1.8 Grammaire finale

<PROGRAM> → <FUNCT-LIST> <PROG-TAIL>
→ <INSTRUCT-LIST>
<PROG-TAIL> → <INSTRUCT-LIST>
→ epsilon
<FUNCT-LIST> → <FUNCT-LIST-BEG> <FUNCT-LIST-END>
<FUNCT-LIST-BEG> → <FUNCT>
<FUNCT-LIST-END> → <FUNCT> <FUNCT-LIST-END>
→ epsilon
<FUNCT> → funct-def id <ARG-LIST> open-brac <INSTRUCT-LIST> close-brac
<ARG-LIST> → open-par <ARG-LIST-BEG> <ARG-LIST-END> close-par
<ARG-LIST-BEG> → variable
→ epsilon
<ARG-LIST-END> → coma variable <ARG-LIST-END>
→ epsilon
<INSTRUCT-LIST> → <INSTRUCT> semicolon <INSTRUCT-LIST>
→ epsilon
<FUNCT-CALL> → funct-name open-par <FUNCT-CALL-ARG> close-par
<FUNCT-CALL-ARG> → <FUNCT-CALL-ARG-BEG> <FUNCT-CALL-ARG-END>
<FUNCT-CALL-ARG-BEG> → <EXP>
→ epsilon
<FUNCT-CALL-ARG-END> → coma <EXP> <FUNCT-CALL-ARG-END>
→ epsilon
<INSTRUCT> → variable equal <EXP>
→ ret <EXP>
→ <COND>
<COND> → open-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
<COND-END> → close-cond open-brac <INSTRUCT-LIST> close-brac
→ add-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
→ epsilon
<SIMPLE-EXP> → <FUNCT-CALL>
→ variable
→ int
→ string
→ open-par <EXP> close-par
<EXP> → <EXP-2> <EXP-TAIL>
<EXP-TAIL> → equiv <EXP-2> <EXP-TAIL>
→ gt <EXP-2> <EXP-TAIL>
→ epsilon
<EXP-2> → <EXP-3> <EXP-2-TAIL>
<EXP-2-TAIL> → add <EXP-3> <EXP-2-TAIL>
→ minus <EXP-3> <EXP-2-TAIL>
→ epsilon
<EXP-3> → <SIMPLE-EXP> <EXP-3-TAIL>
<EXP-3-TAIL> → mul <SIMPLE-EXP> <EXP-3-TAIL>
→ div <SIMPLE-EXP> <EXP-3-TAIL>
→ epsilon

2 Scanner

reconnait plus de token que ceux supporté par la grammaire.
because passé de grammaire complète à grammaire simplifié

3 Parser

3.1 Table de parsing

3.2 First and follow