INFO-F403 Introduction to Language Theory and Compilation

Chapeaux Thomas
Dagnely Pierre

March 18, 2013

Table des matières

1 Introduction

Le but du projet était de construire un compilateur d'une version simplifiée de Perl en ASM/ARM devant tourner dans une architecture Android. Nous avons choisi d'implémenter notre compilateur en Python 2.7[1].

La première partie de ce rapport se concentrera sur l'analyse du langage, 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 les outils que nous avons implémentés (scanner et

[1] http://www.python.org/

parser) qui transforment un fichier .perl en un AST et finalement, nous expliquerons la génération du code ASM/ARM depuis celui-ci.

2 La grammaire

2.1 Définition des unités lexicales

La première étape de ce projet consiste à récupérer et analyser toutes les unités lexicales du langage, c'est à dire tous les "composants" possibles.
Pour cette partie nous nous sommes basés sur la grammaire complète donnée au début du projet.
Le scanner utilise aussi cette grammaire, mais les étapes suivantes ne se basent plus que sur la grammaire simplifiée, ainsi certaines unités lexicales ne sont pas supportés par notre compilateur : seul ceux cités dans la grammaire le sont.

unité lexicale	expression régulière
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

unité lexicale	expression régulière
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

2.2 Modification

Nous nous sommes basés sur la grammaire simplifiée afin de faciliter certaines étapes du projet, la grammaire complète présentant certaines difficulté. Nous avons ajoutés la grammaire simplifiée initiale en annexe 5.3, ainsi ne sont reprise ci-dessous que les règles après modifications.
Mais un numéro de ligne indique à chaque fois quelle règle a été modifiée.

Pour rendre cette grammaire LL(1), nous avons suivi plusieurs étapes :

2.2.1 Suppression des symboles inutiles

On doit commencer par retirer tous les symboles non-productifs et tous les symboles inaccessibles.

Ici tous les symboles sont utiles, on ne modifie donc pas la grammaire

2.2.2 Gestion des priorités et associativité

On veut retirer les ambiguïtés liées aux priorités et à l'associativité.
Cela ne concerne que la règle EXP (29-36), 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>

2.2.3 left factoring

On modifie toute règle ayant un même premier symbole généré.
Cela ne concerne que les règles EXP, EXP-2, EXP-3 et PROGRAM(1-3) :
Cependant cette étape impacte l'associativité gauche des opérations, afin de résoudre ce problème de la manière la plus simple, nous avons décidé d'imposer l'utilisation des parenthèses dans les calculs.

<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

2.2.4 Left recursion

Les grammaires LL(k) ne peuvent contenir de règles du type $A \to A\beta$.
Cela concerne FUNCT-LIST (4-5), ARG-LIST(8-10), INSTRUCT-LIST(11-12) et FUNCT-CALL-ARG(14-16) :

<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

2.2.5 Suppression des productions unitaires

La règle FUNCT-ARG (7) est dans ce cas, sa suppression n'est pas obligatoire, mais elle n'apporte rien à la grammaire et nous préférons donc la retirer.
On la fusionne donc avec ARG-LIST.

2.2.6 Modifications

La grammaire ainsi produite est donc LL(1). Mais nous avons également pratiquées quelques modifications afin de simplifier la grammaire et parfois éviter certaines ambiguïtés.

Nous avons :

1. Ajouter la fonction Perl print, afin de produire du code ASM plus facilement utilisable.

2. Supprimer la possibilité de faire des divisions.
   En effet celle-ci ne sont pas nativement gérées en assembleur par les processeurs ARM.
   Il nous semblait donc préférable de ne pas les implémenter afin de simplifier cette partie et nous concentrer sur le compilateur, le vrai but de ce projet. Nous les avons néanmoins laissés dans la grammaire, mais elle ne sont pas gérée par le générateur de code

2.2.7 Grammaire finale

0) <S> → <PROGRAM> end-symbol[2]
1) <PROGRAM> → <FUNCT-LIST> <PROG-TAIL>
2) → <INSTRUCT-LIST>
3) <PROG-TAIL> → <INSTRUCT-LIST>
4) <FUNCT-LIST> → <FUNCT-LIST-BEG> <FUNCT-LIST-END>
5) <FUNCT-LIST-BEG> → <FUNCT>
6) <FUNCT-LIST-END> → <FUNCT> <FUNCT-LIST-END>
7) → epsilon
8) <FUNCT> → funct-def id <ARG-LIST> open-brac <INSTRUCT-LIST> close-brac
9) <ARG-LIST> → open-par <ARG-LIST-BEG> <ARG-LIST-END> close-par
10) <ARG-LIST-BEG> → variable
11) → epsilon
12) <ARG-LIST-END> → coma variable <ARG-LIST-END>
13) → epsilon
14) <INSTRUCT-LIST> → <INSTRUCT> semicolon <INSTRUCT-LIST>
15) → epsilon
16) <FUNCT-CALL> → funct-name open-par <FUNCT-CALL-ARG> close-par
17) → perl-prin open-par <FUNCT-CALL-ARG> close-par
18) <FUNCT-CALL-ARG> → <FUNCT-CALL-ARG-BEG> <FUNCT-CALL-ARG-END>
19) <FUNCT-CALL-ARG-BEG> → <EXP>
20) → epsilon
21) <FUNCT-CALL-ARG-END> → coma <EXP> <FUNCT-CALL-ARG-END>
22) → epsilon
23) <INSTRUCT> → variable equal <EXP>
24) → ret <EXP>
25) → <COND>
26) → <FUNCT-CALL>
27) <COND> → open-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
28) <COND-END> → close-cond open-brac <INSTRUCT-LIST> close-brac
29) → add-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
30) → epsilon
31) <SIMPLE-EXP> → <FUNCT-CALL>
32) → variable
33) → int
34) → string
35) → open-par <EXP> close-par
36) <EXP> → <EXP-2> <EXP-TAIL>
37) <EXP-TAIL> → equiv <EXP-2>
38) → gt <EXP-2>
39) → epsilon
40) <EXP-2> → <EXP-3> <EXP-2-TAIL>
41) <EXP-2-TAIL> → add <EXP-3>
42) → minus <EXP-3>
43) → epsilon
44) <EXP-3> → <SIMPLE-EXP> <EXP-3-TAIL>
45) <EXP-3-TAIL> → mul <SIMPLE-EXP>
46) → div <SIMPLE-EXP>
47) → epsilon

Attention, bien préciser que nos modifications font qu'on force certaines choses pour pouvoir retirer des ambiguités et donc parser en LL(1).

2.3 Implémentation

Afin de représenter la grammaire dans notre code, Nous avons utilisé une classe `CFGrammar`[3] contenant :

- `symbols` : la liste des symboles

[3]En effet, les grammaires LL(k) sont des CFG

- `terminals` : la liste des terminaux
- `startSymbol` : Le symbole de départ
- `emptySymbol` : Le symbole vide
- `rules` : La liste des règles. Chaque règle $A \rightarrow \alpha_1 ... \alpha_n$ est représentée sous la forme d'une liste de `string`, le premier représentant A et les n suivants représentants les α_i. La grammaire étant une CFG, il n'y a pas d'ambiguité possible.

2.4 Les ensembles First et Follow et la table d'action

Une fois la grammaire implémentée en une instance de `CFGrammar`, il est possible de générer automatiquement les structures nécessaires au parsing LL(k) : L'ensemble $First_k$, l'ensemble $Follow_k$ et la table d'action pour un k donné.

Vu que notre parser était LL(1), seules les méthodes permettant de générer ces structures pour $k = 1$ ont étés implémentées. Celles-ci sont reprises à l'annexe 5.1.

Les tables obtenues sont les suivantes :

Règle	First	Follow
S	funct-name, perl-prin, variable, ret, open-cond, epsilon, end-symbol, funct-def	
FUNCT-CALL-ARG	epsilon, coma, funct-name, perl-prin, variable, int, string, open-par	close-par
PROG-TAIL	funct-name, perl-prin, variable, ret, open-cond, epsilon	end-symbol
PROGRAM	funct-name, perl-prin, variable, ret, open-cond, epsilon, funct-defF	end-symbol
COND	open-cond	semicolon
EXP	funct-name, perl-prin, variable, int, string, open-par	coma, close-par, semicolon, open-brac
FUNCT-CALL-ARG-BEG	epsilon, funct-name, perl-prin, variable, int, string, open-par	coma, close-par
FUNCT	funct-def	funct-def, funct-name, perl-prin, variable, ret, open-cond, end-symbol
FUNCT-CALL	funct-name, perl-prin	semicolon, mul, div, add, minus, equiv, gt, coma, close-par, open-brac
EXP-TAIL	equiv, gt, epsilon	coma, close-par, semicolon, open-brac
ARG-LIST-BEG	variable, epsilon	coma, close-par
FUNCT-LIST	funct-def	funct-name, perl-prin, variable, ret, open-cond, end-symbol
FUNCT-LIST-END	funct-def, epsilon	funct-name, perl-prin, variable, ret, open-cond, end-symbol
EXP-3	funct-name, perl-prin, variable , int, string, open-par	add, minus, equiv, gt, coma, close-par, semicolon, open-brac
EXP-2	funct-name, perl-prin, variable, int, string, open-par	equiv, gt, coma, close-par, semicolon, open-brac
ARG-LIST-END	coma, epsilon	close-par
SIMPLE-EXP	funct-name, perl-prin, variable, int, string, open-par	mul, div, add, minus, equiv, gt, coma, close-par, semicolon, open-brac
ARG-LIST	open-par	open-brac
INSTRUCT	funct-name, perl-prin, variable, ret, open-cond	semicolon
FUNCT-CALL-ARG-END	coma, epsilon	close-apr
EXP-3-TAIL	mul, div, epsilon	add, minus, equiv, gt, coma, close-par, semicolon, open-brac
COND-END FUNCT-LIST-BEG	close-cond, add-cond, epsilon funct-def	semicolon funct-def, funct-name, perl-prin, variable, ret, open-cond, end-symbol
EXP-2-TAIL	add, minus, epsilon	equiv, gt, coma, close-par, semicolon, open-brac
INSTRUCT-LIST	funct-name, perl-prin, variable, ret, open-cond, epsilon	end-symbol, close-brac

	equal	end-symbol	variable	mul	div	minus	semicolon	funct-def	open-cond	gt	string
EXP-3			44								44
PROG-TAIL		3	3						3		
PROGRAM		2	2					1	2		
COND									27		
EXP			36								36
FUNCT-LIST-BEG								5			
FUNCT								8			
FUNCT-CALL											
EXP-TAIL							39			38	
ARG-LIST-BEG			10								
FUNCT-LIST								4			
FUNCT-LIST-END		7	7					6	7		
EXP-2			40								40
ARG-LIST-END											
EXP-3-TAIL				45	46	47	47			47	
ARG-LIST											
INSTRUCT			24						26		
FUNCT-CALL-ARG-END											
SIMPLE-EXP			32								34
COND-END							30				
S		0	0					0	0		
FUNCT-CALL-ARG-BEG			19								19
FUNCT-CALL-ARG			18								18
EXP-2-TAIL						42	43			43	
INSTRUCT-LIST		15	14						14		

	close-brac	add-cond	add	ret	coma	id	int	epsilon	close-par	open-par	close-cond
EXP-3							44			44	
PROG-TAIL				3							
PROGRAM				2							
COND											
EXP							36			36	
FUNCT-LIST-BEG											
FUNCT											
FUNCT-CALL											
EXP-TAIL					39				39		
ARG-LIST-BEG					11				11		
FUNCT-LIST											
FUNCT-LIST-END				7							
EXP-2							40			40	
ARG-LIST-END					12				13		
EXP-3-TAIL			47		47				47		
ARG-LIST										9	
INSTRUCT				25							
FUNCT-CALL-ARG-END					21				22		
SIMPLE-EXP							33			35	
COND-END		29									28
S				0							
FUNCT-CALL-ARG-BEG					20		19		20	19	
FUNCT-CALL-ARG					18		18		18	18	
EXP-2-TAIL			41		43				43		
INSTRUCT-LIST	15			14							

	open-brac	perl-prin	funct-name	equiv
EXP-3		44	44	
PROG-TAIL		3	3	
PROGRAM		2	2	
COND				
EXP		36	36	
FUNCT-LIST-BEG				
FUNCT				
FUNCT-CALL		17	16	
EXP-TAIL	39			37
ARG-LIST-BEG				
FUNCT-LIST				
FUNCT-LIST-END		7	7	
EXP-2		40	40	
ARG-LIST-END				
EXP-3-TAIL	47			47
ARG-LIST				
INSTRUCT		23	23	
FUNCT-CALL-ARG-END				
SIMPLE-EXP		31	31	
COND-END				
S		0	0	
FUNCT-CALL-ARG-BEG		19	19	
FUNCT-CALL-ARG		18	18	
EXP-2-TAIL	43			43
INSTRUCT-LIST		14	14	

3 Traitement des fichiers d'entrées

3.1 Le scanner

Le scanner se base sur la grammaire complète donnée au début du projet, il reconnaît donc plus de token que ceux effectivement géré par le parser, qui lui se base sur la grammaire simplifiée.

3.1.1 Automates fini déterministe

Pour créer le scanner, nous définissons d'abord un DFA représentant tous les tokens du langage.
Ce DFA servira donc de base au scanner.

Représenter le DFA en entier prenant trop de place, nous l'avons séparé en deux entités. Mais l'état de départ de ces deux automates est en fait l'état de départ du DFA complet.

Le premier DFA comprend tous les tokens "non alphabétiques", il reprend tous les caractères spéciaux et les nombres.
Le deuxième DFA comprend tous les tokens "alphabétiques".

La plupart des états pointant vers l'état "ID", nous n'avons représenter ces flèches que par une petite flèche bleue afin de simplifier la lisibilité de l'automate.
Mais il faut à chaque fois voir cela comme une flèche envoyant vers "ID" dés qu'on lit un caractère autres que ceux sur les labels des autres flèches.

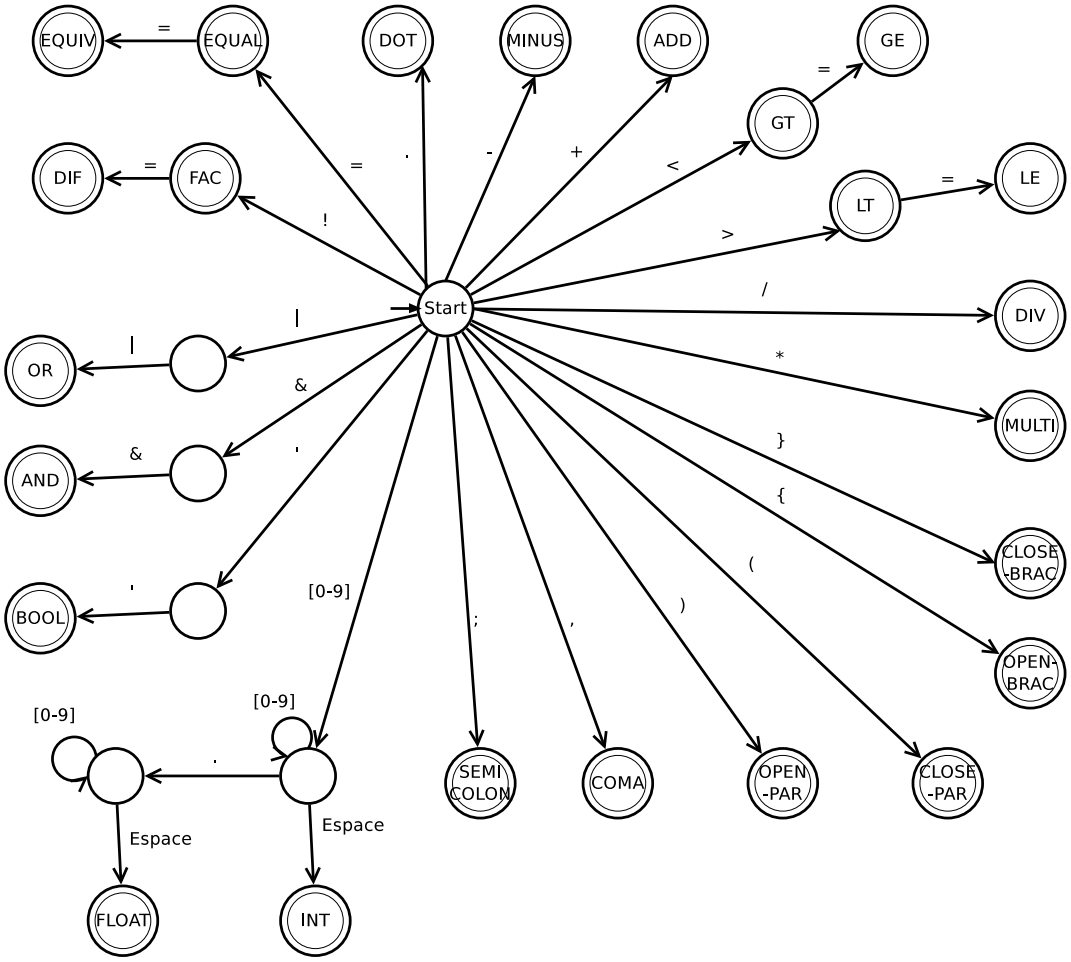

Figure 1: automate "non alphabétique"

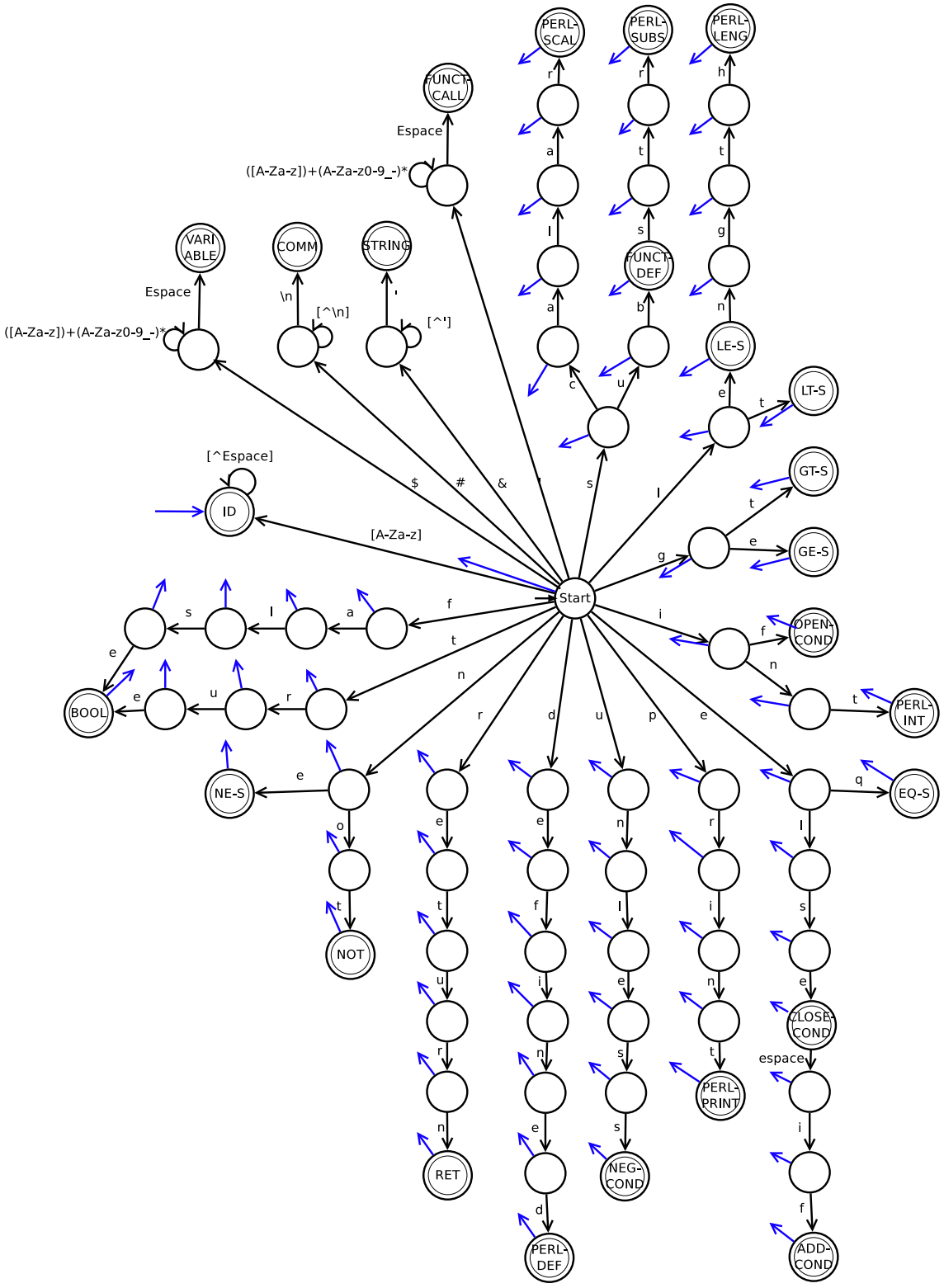

Figure 2: automate "alphabétique"

3.1.2 Implémentation

Nous définissons une méthode `scans` (de la classe `PerlScanner`) qui reçoit le fichier Perl en paramètre. Celui-ci est alors chargé en mémoire et on appelle la fonction `getNextToken`
Celle-ci reçoit un string en paramètre, puis utilise les regex définies dans les DFA pour reconnaître le premier token du string.
Elle renvoi alors le token (une instance de la classe `Token`) et le string duquel elle a retiré le token.

le scanner appelle donc `getNextToken` jusqu'à ce que les strings représentant le document soient vide.

Le scanner étant créé en Python, il est très facile de gérer les regex, en utilisant du code comme celui-ci :

```
1 if line[0] == "g":
2   if re.match("gt[^a-zA-Z0-9__-]", line):
3     line = line[2:]
4     return token.token("GT-S", ""), line
5   if re.match("ge[^a-zA-Z0-9__-]", line):
6     line = line[2:]
7     return token.token("GE-S", ""), line
```

Figure 3: Extrait de la méthode getNextToken()

3.2 Le parser

Le scanner (l'analyse lexicale) renvoie une liste de lexèmes. Le rôle du parser est alors de vérifier que cette liste respecte la grammaire tout en l'arrangeant en une structure ordonnée (le *parsetree*) basée sur la grammaire. Le parser est du type LL(1), ce qui signifie qu'il part du symbole de départ (ici : `S`) qu'il va transformer itérativement à l'aide de la table d'action et d'un symbole de look-ahead jusqu'à obtenir tous les terminaux de l'input.

Notre parser est implémenté en une instance de notre classe `LL1Parser` dont l'attribut `grammar` est une instance de `CFGrammar` représentant la grammaire décrite précédemment. Lors du parsing, il maintient deux structures de données : une liste `output` décrivant les actions effectuées à chaque étape (`M` pour un matching, `Pi` pour l'application de la règle i, `A` si l'input est accepté et `E` si l'input est refusé) et un arbre `parseTree` structurant les tokens. Celui-ci est composé de `ParseTreeNode`, contenant une référence vers un token (`value`), une référence vers le noeud père (`father`) et une liste ordonnée de noeuds enfants (`children`). Le parser possède donc également une référence `currentNode` vers le noeud actuel dans le parse tree.

Le détail de l'implémentation est donné à l'annexe 5.2. Celle-ci se base sur une fonction récursive qui récupère le noeud actuel dans le parse tree et le premier symbole de l'input restant, puis décide de la règle (ou du matching) à appliquer sur base de la table d'action. Dans le cas où on applique une règle, la fonction s'appelle elle-même pour chaque symbole produit par la règle.

3.3 Transformation en AST

Étant donné que la grammaire contient de nombreuses règles superflues pour permettre au parser de n'utiliser qu'un seul symbole de look-ahead, le parse tree renvoyé par le parser contient également beaucoup d'informations superflues. Avant de l'utiliser pour générer du code, nous le simplifions donc en un AST (Abstract Syntax Tree) ne contenant que les informations nécessaires à la génération du code.

Cette étape, effectuée par notre classe `SyntaxTreeAbstracter` prenant un parse tree en entrée, ressemble fort à une inversion des modifications décrites à la section 2.2 et consiste majoritairement en de la manipulation des noeuds de l'arbre. Chaque méthode gère un type de noeud particulier et se charge d'appeller la méthode correspondant aux type des fils du noeud donné en paramètre. La figure 4 montre deux exemples de ces méthodes.

```
def abstractFctCall(self, fctCallNode):
  nameNode = fctCallNode.children[0]
  name = nameNode.value.name
  if (name == "FUNCT_NAME"):
    name = nameNode.value.value
  abstractFctCallNode = parseTreeNode(token.token("Fct_Call", value=name))
  # children are the arguments
  for argRoot in fctCallNode.findToken("FUNCT_CALL_ARG", maxDepth=2):
    for firstArgNode in argRoot.findToken("FUNCT_CALL_ARG_BEG"):
      expNode = firstArgNode.children[0]
      abstractFctCallNode.giveNodeChild(self.abstractExp(expNode))
    for nextArgNode in filterargRoot.findToken("FUNCT_CALL_ARG_END"):
      expNode = nextArgNode.children[1]
      abstractFctCallNode.giveNodeChild(self.abstractExp(expNode))
  return abstractFctCallNode

def abstractSimpleExp(self, simpleExpNode):
  simpleExpTypeNode = simpleExpNode.children[0]
  simpleExpType = simpleExpTypeNode.value.name
  if (simpleExpType == "INT" or simpleExpType == "STRING" or simpleExpType ==
       "VARIABLE"):
    return parseTreeNode(token(simpleExpType, value=simpleExpTypeNode.value.
        value))
  elif (simpleExpType == "FUNCT_CALL"):
    return self.abstractFctCall(simpleExpTypeNode)
  elif (simpleExpType == "OPEN_PAR"):
    return self.abstractExp(simpleExpNode.children[1])
  else:
    raise Exception("Unknown Simple Expression Type : " + str(simpleExpType))
```

Figure 4: Exemples de méthodes du SyntaxTreeAbstracter

3.4 Génération du code

3.4.1 Implémentation

La génération du code consiste à prendre un AST et à le parcourir. En fonction des nœuds rencontrés on écrira alors certaines instructions ASM dans le fichier d'output, en traduisant les instructions représentées dans l'AST en code assembleur correspondant.

Le fonctionnement est assez similaire à celui du `SyntaxTreeAbstracter`, on reçoit un AST qu'on parcours récursivement et en fonction du type de nœud rencontré on appellera une fonction le traduisant en code assembleur.
Nous définissons deux strings, `code` et `header` qui contiendront le future code assembleur et seront concaténé à la fin.
Nous mettons dans `header` les données ASM nécessaire au lancement du programme et nous y définissons en global tous les strings rencontrés au cours du parcours de l'AST (la partie `.data` donc).
`code` contiendra donc toutes les instructions assembleurs, d'abord les éventuelles fonctions puis le code du main (la partie `.text`).

Le code en lui même est assez simple et suit celui de l'exemple ci-dessous. La vrai difficulté de cette partie était de trouver comment traduire en assembleur les différentes instructions, ce langage étant par nature limité. Nous avons de plus décidé de ne pas importer de fonctions C dans notre code afin de ne pas risquer de problème de librairies manquantes et surtout de conserver uniquement du code assembleur, donc plus optimisé et rapide.

```
1  def instruct_list(self, codeNode):
2    for child in codeNode.children:
3      if child.value.name =="Cond":
4        self.cond(child)
5      elif child.value.name =="Assign":
6        self.assign(child)
7      elif child.value.name =="return":
8        self.retur(child)
9      elif child.value.name =="Fct_Call":
10       self.funct_call(child)
11     elif child.value.name !="Instr" and child.value.value !="END":
12       raise "instruct_list non valide"
13     self.code = self.code +"\n"
14
15 def funct_call(self, codeNode):
16   if codeNode.value.value == "PERL_PRIN":
17     for stringNode in codeNode.children:
18       if stringNode.value.name == "STRING":
19         self.registerString(stringNode.value.value)
20         self.code = self.code + " /* syscall write  */ \n"
21         self.code = self.code + " MOV   R0, #1\n"
22         self.code = self.code + " LDR   R1, ="+self.listString[stringNode.
               value.value]+"\n"
23         self.code = self.code + " LDR   R2, ="+self.listStringLen[
               stringNode.value.value]+"\n"
24         self.code = self.code + " MOV   R7, #4\n"
25         self.code = self.code + " SWI   #0\n"
26       else:
27         raise "perl_print ne prend que des strings en parametre"
28
29   else: # fonctions definies par l utilisateur
```

Figure 5: Extrait de code du générateur de code

3.4.2 Traductions des instructions en code ASM

On défini une série de fonctions traduisant un instruction spécifique en code assembleur, souvent ces fonctions s'appellent mutuellement entre-elles.

Les expressions
Ce sont les opérations de bases. On commence par mettre dans deux registres vides les deux paramètres du calculs (ces deux paramètres pouvant nécessité d'être calculé via un appel de fonction, une autre expression, ...). Il suffit alors de faire le calcul correspondant à l'expression sur ces deux registres. On renvoi ensuite le (numéro du) registre contenant le résultat. Puis on vide éventuellement les registres si on les utilise plus.

```
1  MOV   R0, R5
2  BL    metA1
3  MOV   R7, R0
4  MOV   R8, #3
5  ADD   R9, R7, R8
```

Figure 6: Exemple d'expression ($number = &metA1($arg1)+3;) en ASM

Les assignations
On commence par analyser la variable, si elle existe on récupère le registre ou elle est stockée (via un dictionnaire ayant en clé le nom de la variable et en valeur son numéro de registre) sinon on lui attribue un nouveau registre libre. Ensuite on calcule la valeur de l'assignation, ce qui peut nécessiter à nouveau d'autres calculs et codes assembleur. L'assignation se résume alors à mettre dans le registre de la variable la valeur du registre contenant le résultat de cette assignation.

```
1  SUB   R7, R6, R5
2  ADD   R8, R5, R7
3  MUL   R7, R4, R8
4  MOV   R4, R7
```

Figure 7: Exemple d'assignation ($arg1=$arg1*($arg2+($arg3-$arg2));) en ASM

Les returns
Return fonctionne de la même manière, sauf qu'au lieu de mettre le résultat de l'assignation dans le registre d'une variable, on le met dans le registre 0, conçu pour contenir la valeur de retour des fonctions.

```
1  def retur(self, codeNode):
2    for child in codeNode.children:
3      if child.value.name == "OPERATOR":
4        result = self.expression(child)
5        self.code = self.code + " MOV   R0, R"+str(result)+"\n"
6        if result not in self.listVariable.values():
7          self.listRegister[result] = 0
8      elif child.value.name == "STRING":
9        self.registerString(child.value.value)
10       self.code = self.code + " LDR   R0, ="+self.listString[child.value.
             value]+"\n"
11     elif child.value.name == "INT":
12       self.code = self.code + " MOV   R0, #"+child.value.value+"\n"
13     elif child.value.name == "VARIABLE":
14       self.code = self.code + " MOV   R0, R"+str(self.getRegisterOfVariable
             (child.value.value))+"\n"
15     elif child.value.name == "Fct_Call":
16       self.funct_call(child)
17       # le resultat est deja ans R0
18     else:
19       raise Exception("An error occurs during a return")
```

Figure 8: Fonction calculant les returns

Les conditions
L'AST représente les conditions sous forme d'arbres en ajoutant les else/elsif comme étant un fils du if/elsif précédant. Un élément de la condition contient donc comme fils possibles :

1. Une expressions.
   On appelle donc la fonction les analysant.

2. Une liste d'instruction.
   On appelle donc la fonction instruct-list **??**.

3. Un else ou elsif le suivant.
   On appelle donc récursivement la fonction gérant les conditions.

Le principal problème des conditions est qu'elles peuvent être imbriquées. Or en assembleur les conditions sont gérées via des branchements utilisant différents labels. Pour appeler les bons labels on doit donc à chaque fois savoir précisément à quel niveau on se trouve dans l'imbrication des conditions.
On définit deux types de labels :

else*ab* : permet de passer au else/elsif suivant (si la condition du if/elsif courant n'est pas remplie)
a est un nombre identifiant dans qu'elle bloc de condition on se trouve.
b est un nombre identifiant à quel niveau on se trouve dans ce bloc.

end*a* : permet de quitter les conditions. Si on est rentré dans un des if/elsif, à la fin on saute directement à ce point pour éviter les autres elsif/else défini ensuite.
Il n'y a donc besoin que du nombre identifiant dans qu'elle bloc de condition on se trouve, ce label étant commun à tous le niveau de la condition.

```
  MOV   R7, #6
  CMP   R4, R7
  BNE else60
  MOV   R7, #2
  CMP   R5, R7
  BNE else70
  MOV   R7, #3
  CMP   R6, R7
  BNE else80
  /* syscall write  */

  B end8
else80:
  MOV   R7, #3
  CMP   R4, R7
  BNE else81
  /* syscall write  */

  B end8
else81:
  /* syscall write  */

end8:
  B end7
else70:
  /* syscall write  */

end7:
  B end6
else60:
  /* syscall write  */

end6:
```

Figure 9: Exemple de conditions imbriquées en ASM

```
  if $arg1 == 6 {
    if $arg2 == 2 {
      if $arg3 == 3 { print ('passage param : ok'); }
      elsif $arg4 == 3 { print ('passage param : erreur'); }
      else { print ('passage param : erreur'); };
    }
    else { print ('passage param : erreur'); }; }
  else { print ('passage param : erreur'); };
```

Figure 10: Code Perl correspondant au code ASM de la figure 9

les appels de fonctions
Ils peuvent être de deux types :

- Un appel à la fonction perl `print`.
  On utilise alors un system call write linux pour gérer cet affichage

```
1  /* syscall write  */
2  MOV   R0, #1
3  LDR   R1, =str12
4  LDR   R2, =len12
5  MOV   R7, #4
6  SWI   #0
```

Figure 11: Exemple de print en ASM

- Un appel à une fonction définie dans le code.
  Il suffit de mettre dans les registres 0 à 3 les arguments de la fonctions. Et d'écrire dans le code assembleur : "BL <nom de la fonction>".
  Le programme ira alors à la première instruction de la fonction. Puis quand celle-ci sera finie, il reviendra automatiquement à l'instruction suivante dans le code appelant.

  Par manque de temps nous n'avons malheureusement pas eut le temps d'implémenter la gestion de plus de quatre paramètres, cela aurait nécessité d'utiliser le stack, mais nous n'avons pas eut le temps d'implémenter cette partie.

les listes d'instructions et de fonctions
Ces deux parties sont gérées respectivement par `instruct-list` et `funct-list` qui se résument en fait à parcourir ces listes d'instructions ou de fonctions et à appeler les différents méthodes vue ci-dessus.

`funct-list` nécessite cependant une instruction supplémentaire à la fin de chaque fonction. Il faut y rajouter "BX LR" pour quitter la fonction et revenir à l'instruction suivant l'appel de la fonction (cela correspond plus ou moins à "MOV PC, LR").

3.4.3 Variable shadowing

Nous profitons également de ce parcours récursif de l'AST pour vérifier que le code ne viole pas certaines règles. En particulier nous vérifions qu'il ne fait pas appel à des variables inaccessibles.

Cette vérification se fait en plusieurs étapes. D'abord à chaque création/assignation de variable nous l'ajoutons dans une liste les reprenant toutes, ainsi lors d'un appel nous pouvons aisément vérifier si cette variable a déjà été créée, sinon un message d'erreur apparaîtra.
Ensuite nous sommes particulièrement attentif aux variables utilisées dans les fonctions, afin d'éviter qu'elles n'utilisent des variables du main. Pour cela à chaque appel de fonction, on "push" la liste des variables et on en crée un nouvelle qui ne contient que les variables passées en paramètres.
On peut ainsi vérifier qu'on n'accède qu'aux variables accessibles. Il suffit de "poper" cette liste à la fin pour retrouver le contexte initiale.

Pour être complet, il aurait fallu se baser sur le même principe pour gérer les conditions en créant "plusieurs couches" de stacks (par exemple au moyen d'une liste contenant des dictionnaires de variables) :

- A chaque entrée dans une condition : On crée une couche (i.e on ajoute une liste dans la liste représentant le stack)
- A chaque variable créée dans la condition : On l'ajoute à la dernière couche (i.e on l'ajoute dans la dernière liste)
- En sortant d'une condition : On supprime le dernière couche (i.e On supprime la dernière liste)

Rem : On utilise des dictionnaires afin de coupler cela avec la position des variables pour simplifier notre programme. En effet il est très pratique d'utiliser un dictionnaire ayant pour clé la variable et pour valeur son numéro de registre. On peut ainsi vérifier la porté des variables et leur localisation en même temps.

Ce système permettrait aussi de gérer les variables globales, qu'on pourrait imaginer être définie dans le code ou avant les listes de fonctions ou instructions. Il suffirait de leur réserver le premier dictionnaire de notre liste de dictionnaire. Ainsi elle seraient accessibles à tout endroit du code, mais vu qu'on parcours la liste par sa fin pour trouver les variables, elles ne seraient trouvées qu'en dernier. En cas de conflit de nom, ce serait donc la variable locale qui serait trouvée en première.

Malheureusement nous n'avons pas eut le temps d'implémenter cette partie.

3.4.4 Type checking

Sur le même principe, nous vérifions que :

TO DO

3.4.5 Gestion du stack ASM

Par manque de temps nous n'avons pas eut le temps d'implémenter cette partie, ce qui entraîne certaines limitations :

- On ne peut utiliser plus de 8 variables en parallèles.
- On ne peut passer plus de 4 arguments à une fonction.

Avec plus de temps, il aurait été possible de gérer cela assez finement. Il aurait suffit de représenter l'état du stack de chaque registre avec une liste (éventuellement toutes regroupées elle-même dans une liste pour plus de facilité.). Lorsqu'on cherche une variable qui ne se trouve pas dans un des registres, on la cherche alors dans les stack de ces différents registres.
On parcours alors les listes jusqu'à trouver cette variables. Une fois trouvée, on "pop" le stack petit à petit, c'est-à-dire qu'on pop la dernière variable du stack, qu'on la déplace dans un registre vide, qu'on pop la variable suivantes, ... jusqu'à ce que la variable qui nous intéresse sorte du stack et se retrouve dans le registre. Permettant ainsi de l'utiliser.

Inversement quand on veut un registre vide et qu'ils sont tous occupés, il suffirait de pusher un des registres pour le libérer, la variable "perdue" pouvant être aisément retrouvée par la suite, cela ne prête donc pas à conséquence et permet d'utiliser

presque autant de variable que l'on désire (en restant dans les limites matérielles possibles).

Encore une fois nous n'avons malheureusement pas eut le temps de finir cette partie.

4 Guide d'utilisation

Le programme étant écrit en Python il est aisé de le lancer[4].
Une fois dans le dossier, il suffit de lancer la commande :

python compilateur.py -v -i <fichier perl>

L'option -v est optionnelle, si sélectionnée, le programme affichera dans le terminal le résultat de chaque étape intermédiaire.

Il est également possible de lancer chaque partie séparément, elle afficheront alors dans le terminal le résultat.

- python scanner.py -i <fichier perl>
- python parser.py -i <fichier perl>
- python syntaxtreeabstracter.py -i <fichier perl>

Le code assembleur assembleur ainsi produit a été conçu et tester pour être compiler avec le compilateur `android-ndk-r8d` et tester avec `adb`, via ces commandes (une fois l'émulateur lancé, ou un smartphone android connecté).

1. android-ndk-r8d/toolchains/arm-linux-androideabi-4.7/prebuilt/linux-x86/bin/arm-linux-androideabi-as -o program.o program.S
2. android-ndk-r8d/toolchains/arm-linux-androideabi-4.7/prebuilt/linux-x86/bin/arm-linux-androideabi-ld -s -o exécutable program.o
3. adb push exécutable /data/local/tmp/
4. adb shell /data/local/tmp/exécutable

Il suffit de modifier la partie <linux-x86> en fonction du système d'exploitation.

Nous avons également inclus un fichier perl contenant des instructions gérées par notre grammaire, ainsi que le fichier assembleur correspondant généré par notre compilateur, dans le dossier `inputTests`

[4]Il nécessite juste d'installer python 2.7

5 Annexes

5.1 Méthodes générant les tables First, Follow et la table d'action

Comme expliqué précédemment, notre classe `CFGrammar` nous a permis de générer automatiquement les tables $First_1$ et $Follow_1$, ainsi que la table d'action. Voici les méthodes utilisées.

La méthode générant $First_1$ (décrite à la Figure 12) va explorer les applications possibles de règles jusqu'à trouver tous les terminaux obtenables, celle générant $Follow_1$ (Figure 13) recherche, pour chaque symbole A, les éléments de $First_1$ correspondant aux symboles générés après A dans une des règles. Finalement, la dernière méthode (Figure 14) utilise les deux tables précédentes pour déterminer quelle règle appliquer lorsqu'on doit matcher un symbole à un terminal.

```
for A in self.symbols:
  if A in self.terminals:
    first[A] = [A]
stable = False
while not stable:
  stable = True
  for A in self.symbols:
    for rule in (rules beginning with A):
      found_epsilon = True
      current_symbol = 0
      while found_epsilon and current_symbol + 1 < len(rule):
        found_epsilon = False
        current_symbol += 1
        for candidate in first[rule[current_symbol]]:
          if candidate == self.emptySymbol:
            found_epsilon = True
          if candidate not in first[A]:
            stable = False
            first[A].append(candidate)
return first
```

Figure 12: Pseudo-code de la méthode générant la table $First_1$

```
follow = {}
first = self.first_1()
stable = False
while (not stable):
  stable = True
  for rule in self.rules:
    for i, A in enumerate(rule):
      if (i == 0):
        continue
      foundEpsilon = True
      epsilonOffset = 0
      while foundEpsilon:
        foundEpsilon = False
        epsilonOffset += 1
        if i + epsilonOffset >= len(rule):
          candidates = follow[rule[0]]
        else:
          candidates = first[rule[i + epsilonOffset]]
        for candidate in candidates:
          if candidate == self.emptySymbol:
            foundEpsilon = True
          elif candidate not in follow[A]:
            stable = False
            follow[A].append(candidate)
return follow
```

Figure 13: Pseudo-code de la méthode générant la table $Follow_1$

```
first = self.first_1()
follow = self.follow_1()
M = {}
for i, rule in enumerate(self.rules):
  A = rule[0]
  foundEpsilon = True
  current_symbol = 0
  while foundEpsilon:  # will not loop because follow never contains epsilon
    foundEpsilon = False
    current_symbol += 1
    if current_symbol >= len(rule):
      candidates = follow[A]
    else:
      candidates = first[rule[current_symbol]]
    for candidate in candidates:
      if (candidate == self.emptySymbol):
        foundEpsilon = True
      else:
        M[A][candidate] = i
```

Figure 14: Pseudo-code de la méthode générant la table d'action pour $k = 1$

5.2 Implémentation du parser

Notre parser est basé sur l'exemple de parser LL(k) donné dans le syllabus. Notre parser n'utilise pas explicitement de symbole de fin, mais celui-ci est décrit dans la grammaire et ajouté par le scanner. Il est donc traité comme les autres symboles.

```
def parse(self, inputText):
  self.input = inputText  # list of symbols
  self.output = []
  self.error = False
  self.success = False

  self.parseTree = parseTreeNode(token(self.grammar.startSymbol))
  self.currentNode = self.parseTree

  self.parse_recursiveCall()
  return self.output

def parse_recursiveCall(self):
  if (self.error or self.accept):
    return
  stack_token = self.currentNode.value
  input_token = self.input[0]  # only one character as we have an LL1-parser
  # notation from the syllabus - pp. 116
  X = stack_token.name
  u = input_token.name

  if (u in self.M[X]):
    self.produce(self.M[X][u])
  elif (X in self.grammar.terminals and X == u):
    self.match()
  else:
    self.trigger_error(X, u)

def trigger_error(self, X, u):
  self.output.append("E")
  if u not in self.grammar.symbols:
    raise ParseError("Unknown symbol", u)
  else:
    raise ParseError("Misplaced symbol", u)

def trigger_accept(self):
  self.output.append("A")
  self.success = True

def produce(self, i):
  self.output.append("P" + str(i))
  saved_current = self.currentNode
  for produced in (symbols produced by rule i):
    if (produced != self.grammar.emptySymbol):
      self.currentNode.giveChild(token(produced))
      self.currentNode = self.currentNode.children[-1]
      self.parse_recursiveCall()
      self.currentNode = saved_current
  self.currentNode = saved_current

def match(self):
  parseTreeToken = self.currentNode.value
  inputToken = self.input.pop(0)
  parseTreeToken.value = inputToken.value
  self.output.append("M")
  if (len(self.input) == 0):
    self.trigger_accept()
```

Figure 15: Pseudo-code du Parser

5.3 Grammaire initiale

1) <PROGRAM> → <FUNCT-LIST> <INSTRUCT-LIST>
2) → <FUNCT-LIST>
3) → <INSTRUCT-LIST>
4) <FUNCT-LIST> → <FUNCT>
5) → <FUNCT-LIST> <FUNCT>
6) <FUNCT> → funct-def id open-par <FUNCT-ARG> close-par open-brac <INSTRUCT-LIST> close-brac
7) <FUNCT-ARG> → open-par <ARG-LIST> close-par
8) <ARG-LIST> → <ARG-LIST> coma variable
9) → variable
10) → epsilon
11) <INSTRUCT-LIST> → <INSTRUCT-LIST> <INSTRUCT> semicolon
12) → <INSTRUCT> semicolon
13) <FUNCT-CALL> → funct-name <FUNCT-CALL-ARG>
14) <FUNCT-CALL-ARG> → <FUNCT-CALL-ARG> coma <EXP>
15) → <EXP>
16) → epsilon
17) <INSTRUCT> → variable equal <EXP>
18) → <EXP>
19) → ret <EXP>
20) → <COND>
21) <COND> → open-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
22) <COND-END> → add-cond <EXP> open-brac <INSTRUCT-LIST> close-brac <COND-END>
23) → close-cond open-brac <INSTRUCT-LIST> close-brac
24) → epsilon
25) <SIMPLE-EXP> → int
26) → <FUNCT-CALL>
27) → variable
28) → string
29) <EXP> → <SIMPLE-EXP>
30) → open-par <EXP> close-par
31) → <EXP> add <EXP>
32) → <EXP> minus <EXP>
33) → <EXP> multi <EXP>
34) → <EXP> div <EXP>
35) → <EXP> equiv <EXP>
36) → <EXP> gt <EXP>