Université Libre de Bruxelles

INFO-F-403
Introduction to language theory and compiling

Assignement 2 : The parser

Maxime Desclefs - 362626

Julian Schembri - 380446

2015 - 2016

1 Introduction

Dans le cadre du cours d'introduction à la théorie du langage et de compilation (INFO-F-403), il a été demandé d'implémenter et d'écrire un compilateur pour le langage SUPRALGOL^{2016}, une variante du langage Algol68.

La première partie consistait en la création du scanner. Ce scanner analyse et découpe, ensuite, un code SUPRALGOL^{2016} permettant d'identifier chacun des éléments de ce dernier, et leur correspondance dans ce langage. La deuxième partie consistait en la création du parser. Ce parser permet de détecter les erreurs de syntaxe ne respectant pas la grammaire du langage à l'aide de tous les outils mis à sa disposion. Les différents outils seront énoncés dans le paragraphe suivant.

Il fallait, dans un premier temps, transformer la grammaire SUPRALGOL^{2016} afin de la rendre LL(1). Quatre étapes sont nécessaires à cette transformation : la suppression des symboles inutiles, la suppression de la récursivité à gauche, la factorisation des règles, ainsi que la prise en compte de la priorité et l'associativité des opérateurs pour rendre la grammaire non-ambigüe.
Sur base de cette nouvelle grammaire et quelques manipulations, une *action table* a pu être créée, rendant possible l'exécution du parser LL(1) qu'il a été demandé de réaliser.

Etant donné la demande de justificatifs des différentes étapes, l'attention a été portée sur l'implémentation des différents algorithmes en vue de l'obtention des réponses les plus adéquates. Dans la suite de ce rapport, seront expliqué les différentes opérations, ainsi que les résultats ayant pu être obtenus grâce à ces dernières.

2 Grammaire

Une grammaire permet de définir un langage en définissant sa syntaxe. Elle décrit donc l'ensemble des mots adminissibles, que nous appelerons *terminaux*, à ce langage sur un alphabet donné, que nous appelerons *variables*.

2.1 Suppression des variables inutiles

Une variable est inutile si elle ne rempli pas l'une des fonction suivante : être accessible depuis le symbole de départ et produire quelque chose. Dès lors les variables inutiles détectées, elles peuvent être supprimées des différentes listes ainsi que des différentes règles dans lesquelles elles se trouvent.

2.2 Priorité et associativité des opérateurs

Afin de rendre la grammaire non-ambigüe, il est nécessaire d'appliquer les priorités et les associativités possible sur les différents opérateurs de cette dernière.
Dans les règles de la grammaire, les plus importantes, ç-à-d avec une priorité plus élevée, se trouvent les plus haut dans la grammaire. Ceci peut être expliqué par le parcours des règles (parcours de liste type *ArrayList*) ; en effet les premières règles, ç-à-d tout en haut de la liste, sont executées en premier. Il ne reste donc qu'à intervertir, remplacer, décaler, des règles en les montant pour en augmenter la priorité ou bien les descendre dans le cas échéant. Si des opérateurs ont la même priorité, ils sont mis sur une même ligne.

2.3 Suppresion de la récursivité à gauche

La récursivité à gauche est un problème pour le ≪*Top-Down parsing*≫. En effet, cette classe de parseur utilise, comme demandé dans l'énoncé, la ≪*Left-most derivation*≫. Pour pallier à ce problème, la récursivité à gauche va être remplacée par une autre sorte de récursivité. L'introduction d'une nouvelle variable est nécessaire.

2.4 Factorisation à gauche

Pour les «*Top-Down parsing*», chacunes des règles de la grammaire doit être factorisées à gauche. Comme pour le point précédent sur la recursivité, l'introduction d'une nouvelle variable est nécessaire. Cette étape consiste à rassembler les différents terminaux, ayant un préfixe commun (différent de ϵ), d'une variable, dans cette nouvelle variable recemment introduite. Cela enlève une ambigüité et permet au parser de savoir directement quelle règle choisir sans devoir toutes les parcourir.

3 Table d'action

Comme mentionné dans l'introduction, la construction d'une table d'action est nécessaire au bon fonctionnement du parser. Pour chaque symbole, le parseur doit savoir deux choses sur ce dernier : *First(X)* et *Follow(X)* où X est le symbole. La grammaire étant de type LL(1), les différents algorithmes faisant présence d'un ordre de grandeur k se le voient attribué à 1. *First* est donc l'ensemble des premiers terminaux (vecteur d'un seul terminal) rencontré pouvant commencer un mot généré à partir d'une variable. *Follow* s'applique pour les terminaux qui suivent le mot. Sur base de ces deux vecteurs , la table d'action peut être construite. L'action table permet donc au parseur de faire la liaison, entre les variables et les terminaux, en indiquant le numéro de la règle qui permet cette dernière.

3.1 Valeur de First et Follow

Les différentes étapes permettant d'obtenir ces resultats sont proposées en annexe. Ces étapes ont été réalisées sur base des algorithmes présents dans les slides du cours.

	First	**Follow**
Program	*begin*	*eps*
Code	[$VarName$], *print*, *read*, for, *eps*, *while*, if	fi, *od*, *else*, *end*
InstList	[$VarName$], *print*, *read*, for, *while*, if	fi, *od*, *else*, *end*
Instruction	[$VarName$], *print*, *read*, for, *while*, if	fi, *od*, *else*, *end*, ;
InstList'	*eps*, ;	fi, *od*, *else*, *end*
Assign	[$VarName$]	fi, *od*, *else*, *end*, ;

If	if	fi, od, $else$, end, ;
While	$while$	fi, od, $else$, end, ;
For	for	fi, od, $else$, end, ;
Print	$print$	fi, od, $else$, end, ;
Read	$read$	fi, od, $else$, end, ;
ExprArith	$[VarName]$, (, −, $[Number]$	<=, fi, for, do, $while$, od, $else$, by, end, if, $[Number]$, $read$, (,), ∗, +, −, / =, /, $[VarName]$, $print$, ;, to, <, =, >, >=
ExprArith0	$[VarName]$, (, $[Number]$	<=, fi, for, do, $while$, od, $else$, by, end, if, $[Number]$, $read$, (,), ∗, +, −, / =, /, $[VarName]$, $print$, ;, to, <, =, >, >=
ExprArith0'	eps, ∗, /	<=, fi, for, do, $while$, od, $else$, by, end, if, $[Number]$, $read$, (,), ∗, +, −, / =, /, $[VarName]$, $print$, ;, to, <, =, >, >=
Op'	∗, /	<=, fi, for, do, $while$, od, $else$, by, end, if, $[Number]$, $read$, (,), ∗, +, −, / =, /, $[VarName]$, $print$, ;, to, <, =, >, >=
ExprArith1	$[VarName]$, (, $[Number]$	<=, fi, for, do, $while$, od, $else$, by, end, if, $[Number]$, $read$, (,), ∗, +, −, / =, /, $[VarName]$, $print$, ;, to, <, =, >, >=
ExprArith1'	eps, +, −	<=, fi, for, do, $while$, od, $else$, by, end, if, $[Number]$, $read$, (,), ∗, +, −, / =, /, $[VarName]$, $print$, ;, to, <, =, >, >=

Op”	+, −	<=, *fi*, *for*, *do*, *while*, *od*, *else*, *by*, *end*, *if*, [*Number*], *read*, (,), ∗, +, −, / =, /, [*VarName*], *print*, ;, *to*, <, =, >, >=
ExprArith2	[*VarName*], (, [*Number*]	<=, *fi*, *for*, *do*, *while*, *od*, *else*, *by*, *end*, *if*, [*Number*], *read*, (,), ∗, +, −, / =, /, [*VarName*], *print*, ;, *to*, <, =, >, >=
Cond	[*VarName*], *not*, (, −, [*Number*]	*read*, *fi*, *or*, *for*, *then*, *do*, *while*, [*VarName*], *print*, *od*, *else*, *end*, ;, *if*
If’	*fi*, *else*	*fi*, *od*, *else*, *end*, ;
Cond0	[*VarName*], (, −, [*Number*]	*read*, *fi*, *or*, *for*, *then*, *do*, *while*, [*VarName*], *print*, *od*, *else*, *end*, ;, *if*
SimpleCond	[*VarName*], (, −, [*Number*]	*read*, *fi*, *or*, *for*, *then*, *do*, *while*, [*VarName*], *print*, *od*, *else*, *end*, ;, *if*
Cond1	-	*read*, *fi*, *or*, *for*, *then*, *do*, *while*, [*VarName*], *print*, *od*, *else*, *end*, ;, *if*
Cond1’	*eps*	*read*, *fi*, *or*, *for*, *then*, *do*, *while*, [*VarName*], *print*, *od*, *else*, *end*, ;, *if*
BinOp’	*and*	*read*, *fi*, *or*, *for*, *then*, *do*, *while*, [*VarName*], *print*, *od*, *else*, *end*, ;, *if*
Cond2	-	*read*, *fi*, *or*, *for*, *then*, *do*, *while*, [*VarName*], *print*, *od*, *else*, *end*, ;, *if*
Cond2’	*or*, *eps*	*eps*

BinOp"	*or*	$[VarName]$, *not*, *or*, (, −, $[Number]$
Comp	<=, <, =, >, / =, >=	$[VarName]$, (, −, $[Number]$

3.2 Résultat

	begin	*end*	;	[*VarName*]	:=	−	[*Number*]	(	)	*	/	+	*if*	*then*	*fi*	*else*	*not*	*and*	*or*	=	>=	>	<=	<	/=	*while*	*do*	*od*	*for*	*from*	*by*	*to*	*print*	*read*
Program	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Code	-	2	-	3	-	-	-	-	-	-	-	-	3	-	2	2	-	-	-	-	-	-	-	-	-	3	-	2	3	-	-	-	3	3
InstList	-	-	-	4	-	-	-	-	-	-	-	-	4	-	-	-	-	-	-	-	-	-	-	-	-	4	-	-	4	-	-	-	4	4
Instruction	-	-	-	7	-	-	-	-	-	-	-	-	8	-	-	-	-	-	-	-	-	-	-	-	-	9	-	-	10	-	-	-	11	12
InstList'	-	5	6	-	-	-	-	-	-	-	-	-	-	-	5	5	-	-	-	-	-	-	-	-	-	-	-	5	-	-	-	-	-	-
Assign	-	-	-	13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
If	-	-	-	-	-	-	-	-	-	-	-	-	29	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
While	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	50	-	-	-	-	-	-	-	-
For	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	51	-	-	-	-	-
Print	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	52	-
Read	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	53
ExprArith	-	-	-	15	-	14	15	15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ExprArith0	-	-	-	18	-	-	18	18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ExprArith0'	-	17	17	17	-	17	17	17	17	16	16	17	17	-	17	17	-	-	-	17	17	17	17	17	17	17	17	17	17	-	17	17	17	17
Op'	-	-	-	-	-	-	-	-	-	25	26	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ExprArith1	-	-	-	21	-	-	21	21	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ExprArith1'	-	20	20	20	-	19	20	20	20	20	20	19	20	-	20	20	-	-	-	20	20	20	20	20	20	20	20	20	20	-	20	20	20	20
Op"	-	-	-	-	-	28	-	-	-	-	-	27	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ExprArith2	-	-	-	22	-	-	23	24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cond	-	-	-	33	-	33	33	33	-	-	-	-	-	-	-	-	32	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
If'	-	-	-	-	-	-	-	-	-	-	-	-	-	-	30	31	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cond0	-	-	-	34	-	34	34	34	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SimpleCond	-	-	-	41	-	41	41	41	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cond1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cond1'	-	37	37	37	-	-	-	-	-	-	-	-	37	37	37	37	-	36	37	-	-	-	-	-	-	37	37	37	37	-	-	-	37	37
BinOp'	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	42	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cond2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cond2'	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	39	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
BinOp"	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	43	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Comp	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	44	45	46	47	48	49	-	-	-	-	-	-	-	-	-

4 Conclusion

L'implémentation des différents algorithmes n'a pas été une tache facile. Certaines opérations et vérifications ne sont pas, toujours, entièrement décrites dans les slides. Cette implémentation n'est donc sûrement pas parfaite. Plusieurs tests ont été effectués et ont pointé de nombreuses erreurs, qui ont pu être corrigées par la suite. Cependant, pour les recommandations du projet, les résultats ont l'air dans l'ensemble corrects.

Lors de la précédente itération, le rapport avait été quelqu'un peu négligé et cela s'était ressenti. C'est donc pour cela que celui-ci a été mieux préparé, et realisé de la manière la plus claire possible.

5 Annexe