TP 2: Mise en œuvre d'un analyseur syntaxique pour le langage C⋆

1 But de cette séance de travaux pratiques

Le but de cette séance de TP est de réaliser un analyseur syntaxique pour un langage appelé C⋆ qui est un sous-ensemble strict du langage C. Cette séance est l'occasion de mettre en œuvre l'algorithme vu en cours pour la réalisation d'un analyseur syntaxique récursif descendant (de la famille dite LL(1)). Pour ce faire on vous rappelle qu'il convient de calculer trois fonctions (de difficulté de calcul croissante) et qui doivent être calculées dans l'ordre suivant (car chacune dépend de la précédente) :

1. La fonction NULL qui indique si un non-terminal (ou simplement une production, c'est-à-dire la partie droite d'une règle de la grammaire) peut dériver (produire) le mot vide. C'est donc un prédicat, *i.e.* une fonction renvoyant un booléen.
2. La fonction FIRST qui indique les terminaux (ou lexèmes, ou *tokens*) par lesquels peuvent commencer tous les mots produits par un non-terminal (ou simplement une production). Le calcul de cette fonction a besoin des résultats produits par la fonction NULL*.
3. Enfin la fonction FOLLOW qui indique pour un non-terminal (pouvant dériver le mot vide) par quels non-terminaux il peut être suivi.

Ces calculs vous permettront à leur tour de fabriquer une table de prédiction de laquelle se déduira l'algorithme d'analyse syntaxique.

2 Définition de la grammaire du langage C⋆

Nous vous indiquons une grammaire pour le langage C⋆ illustrée par la figure 1 qui utilise des constructions supplémentaires par rapport aux grammaires vues en cours : nous introduisons des opérateurs $?, ^\star, ^+$ qui indiquent respectivement qu'un non-terminal peut apparaître :

1. 0 ou 1 une seule fois.
2. 0 ou un nombre arbitraire de fois.
3. 1 fois ou plus.

Ces constructions devront être dépliées en utilisant des non-terminaux supplémentaires afin de pouvoir mener les calculs simplement. Par ailleurs, nous avons utilisé des récursions à gauche (qui sont interdites pour fabriquer une grammaire LL(1)), de plus de nombreuses règles ont des préfixes communs pour le même non-terminal. Par ailleurs, nous avons simplement indiqué les priorités des opérations de manière informelle. Il faudra vous référer au cours pour transformer cette grammaire en une grammaire potentiellement LL(1).

(1)	**fichier**	→	**programme** SYM_EOF
(2)	**programme**	→	**include*** **def_ou_decl***
(3)	**include**	→	SYM_INCLUDE SYM_STRING
(4)	**def_ou_decl**	→	SYM_EXTERN **decl_fonction**
(5)		→	SYM_EXTERN ? **def_vars**
(6)		→	**def_fonction**
(7)	**decl_fonction**	→	**type_retour** SYM_IDENTIFIER SYM_LPARENTHESIS **parametres** SYM_RPARENTHESIS SYM_SEMICOLON
(8)	**type_retour**	→	SYM_VOID
(9)		→	**type**
(10)	**parametres**	→	**parametre**(SYM_COMMA **parametre**)*
(11)		→	ε
(12)	**parametre**	→	**type** SYM_IDENTIFER
(13)	**type**	→	SYM_UINT64 SYM_ASTERISK ?
(14)	**def_vars**	→	SYM_UINT64 **vars** SYM_SEMICOLON
(15)	**vars**	→	SYM_ASTERISK ? SYM_IDENTIFIER (SYM_COMMA SYM_ASTERISK ? SYM_IDENTIFIER)*
(16)	**def_fonction**	→	**type_retour** SYM_IDENTIFIER SYM_LPARENTHESIS **arguments** SYM_RPARENTHESIS **instruction**
(17)	**instruction**	→	**if_then_else**
(18)		→	**while**
(19)		→	**affectation** SYM_SEMICOLON
(20)		→	**appel** SYM_SEMICOLON
(21)		→	**return** SYM_SEMICOLON
(22)		→	**bloc**
(23)	**if_then_else**	→	SYM_IF SYM_LPARENTHESIS **condition** SYM_RPARENTHESIS **instruction** **else** ?
(24)	**else**	→	SYM_ELSE **instruction**
(25)	**while**	→	SYM_WHILE SYM_LPARENTHESIS **condition** SYM_RPARENTHESIS **instruction**
(26)	**affectation**	→	SYM_ASTERISK **dereferencement** SYM_ASSIGN **expression**
(27)		→	SYM_IDENTIFIER SYM_ASSIGN **expression**
(28)	**dereferencement**	→	SYM_IDENTIFIER
(29)		→	SYM_LPARENTHESIS **expression** SYM_RPARENTHESIS
(30)	**appel**	→	SYM_IDENTIFER SYM_LPARENTHESIS **arguments** SYM_RPARENTHESIS
(31)	**arguments**	→	**expression** (SYM_COMMA **expression**)*
(32)		→	ε
(33)	**condition**	→	**expression op_test expression**
(34)	**op_test**	→	SYM_EQUALITY
(35)		→	SYM_LT
(36)		→	SYM_LEQ
(37)		→	SYM_GT
(38)		→	SYM_GEQ
(39)		→	SYM_NOTEQ
(40)	**return**	→	SYM_RETURN **expression** ?
(41)	**bloc**	→	SYM_LBRACE **def_vars*** **instruction*** SYM_RBRACE
(42)	**expression**	→	
Priorité 1			
(43)		→	**expression** SYM_PLUS **expression**
(44)		→	**expression** SYM_MINUS **expression**
Priorité 2			
(45)		→	**expression** SYM_ASTERISK **expression**
(46)		→	**expression** SYM_DIV **expression**
(47)		→	**expression** SYM_MOD **expression**
Priorité 3			
(48)		→	SYM_LPARENTHESIS **expression** SYM_RPARENTHESIS
(49)		→	SYM_IDENTIFER
(50)		→	SYM_INTEGER
(51)		→	SYM_MINUS **expression**
(52)		→	**cast expression**
(53)		→	SYM_ASTERISK **dereferencement**
(54)		→	**appel**
(55)		→	SYM_STRING
(56)		→	SYM_CHAR
(57)	**cast**	→	SYM_LPARENTHESIS **type** SYM_RPARENTHESIS **expression**

FIGURE 1 – La grammaire du langage C$\star$.

Aide

Nous vous engageons à introduire les non-terminaux suivants :

- **fichier** : un fichier contenant un programme et se terminant par le symbole de fin de fichier. Il s'agira de l'axiome de la grammaire.
- **programme** : un programme qui peut contenir des inclusions d'entête suivis de définitions et de déclarations de variables de fonctions et de variables globales entremêlées (possiblement vide)
- **includes** : des inclusions de fichiers d'entête (possiblement vide).
- **include** : une inclusion de fichier d'entête.
- **defs_ou_decls** : des définitions et des déclarations de variables de fonctions et de variables globales entremêlées (possiblement vide).
- **def_ou_decl** : une unique déclaration ou définition de fonction ou de variable globale.
- **declaration ?** : la présence du mot-clé `extern`.
- **vars_ou_foncs** : un début de définition ou de déclaration de fonction ou de variable jusqu'à l'apparition d'un identifiant (SYM_IDENTIFER). Cette règle se poursuit par l'appel de la règle suivante.
- **vars_ou_foncs_suite** : Cette règle distingue les définitions (et déclarations) de fonctions de celles de variables.
- **autres_vars** : la suite de déclaration de variable quand il y en a déjà une.
- **parametres** : les paramètres d'une fonction (il peut n'y en avoir aucun).
- **autres_parametres** : la suite de paramètres de fonction, quand il y en a déjà un.
- **parametre** : un unique paramètre de fonction.
- **decl_ou_instruction** : le début d'une déclaration de variable ou d'une instruction.
- **pointeur ?** : la présence du symbole `*`.
- **instruction** : les différentes instructions possibles en C⋆.
- **if_then_else** : l'instruction `if then else`.
- **else ?** : la présence d'un `else`.
- **while** : l'instruction `while`.
- **affectation_ou_appel** : le début de l'affectation d'une variable ou d'un appel de fonction.
- **variable_ou_appel** :
- **dereferencement** : le début du déréférencement d'une variable.
- **affectation** : l'opération d'affectation.
- **condition** : une condition.
- **op_test** : les opérateurs de test.
- **return** : l'instruction `return`.
- **valeur_retour** : les valeurs de retour possibles pour l'instruction `return`.
- **bloc** : les blocs d'instructions.
- **vars_decls** : les déclarations de variables locales à un bloc.
- **vars_decl** : une définition de variables locale.
- **instructions** : une suite d'instructions.
- **expression** : le traitement des suites d'opérations de priorité 1.
- **autres_termes** : une suite d'éléments d'opération de priorité 1.
- **terme** : le traitement des suite d'opérations de priorité 2.
- **autres_facteurs** : une suite d'élément d'opération de priorité 2.
- **facteur** : les éléments d'opération de priorité 3.
- **parenthese_ou_cast** : permet de différencier les parenthèses, des casts.
- **var_ou_appel** : le traitement des variables ou des appels de fonction dans les expressions.
- **arguments ?** : teste la présence d'arguments.
- **arguments** : les arguments passés à un appel de fonction.
- **autres_arguments** : la suite d'arguments quand il y en déjà un.

3 Calcul du prédicat Null

Vous calculerez le prédicat NULL pour chacun des non-terminaux de la grammaire. Vous pourrez remplir la table 2 pour récapituler vos résultats. Pour rappel, il suffit de déterminer pour chacun des non-terminaux s'il peut produire le mot vide. Pour cela, on peut utiliser les règles suivantes :

- le prédicat vaut VRAI s'il existe directement une règle qui renvoie le mot vide.
- le prédicat vaut FAUX si chacune des productions associées au non-terminal contient un lexème.
- Dans tous les autres cas, il faut calculer le prédicat NULL pour chacun des non-terminaux qui

composent les productions au non-terminal.

En cas de doute, merci de vous reporter au cours.

Non terminal	Valeur du prédicat Null	Explication
programme		
includes		
include		
defs_ou_decls		
def_ou_decl		
declaration ?		
vars_ou_foncs		
vars_ou_foncs_suite		
autres_vars		
parametres		
autres_parametres		
parametre		
decl_ou_instruction		
pointeur ?		
instruction		
if_then_else		
else ?		
while		
affectation_ou_appel		
variable_ou_appel		
deferencement		
affectation		
condition		
op_test		
valeur_retour		
bloc		
vars_decls		
vars_decl		
instructions		
expression		
autres_termes		
terme		
autres_facteurs		
facteur		
parenthese_ou_cast		
var_ou_appel		
arguments ?		
arguments		
autres_arguments		

Figure 2 – La valeur du prédicat Null pour chacun des non-terminaux de la grammaire de C⋆.

4 Calcul de la fonction First

Vous devez à présent calculer la fonction First. Pour rappel cette fonction indique pour chaque non-terminal (disons X) les lexèmes qui débutent l'ensemble des mots qui peuvent être dérivés à partir de X. Pour cela, il suffit pour chacune des règles associées à X (on considère ici une règle : $X \to \omega$) de déterminer les non-terminaux qui peuvent débuter la production. Ici on va calculer First(ω)). Cette valeur est simplement :

- le lexème qui débute la production ω lorsque précisément celle-ci commence par un lexème ;
- lorsque la production ω commence par une variable (par exemple $\omega = A\omega\prime$), la valeur de First(A). Dans ce cas, il ne faut pas oublier de considérer l'ensemble des variables qui figurent au début de la production si elles peuvent produire le mot vide (ceci a été calculé à l'étape précédente) jusqu'à parvenir sur une variable qui ne produise pas le mot vide, ou bien un lexème. Il faut dans ce cas faire l'union des valeurs de la fonction First pour chacune de ces variables/lexème.

On fait ensuite l'union de tous les lexèmes ainsi obtenus pour chacune des règles associées au non-terminal considéré. Remplissez le tableau 3.

Non terminal	Valeur de la fonction First	Explication
programme		
includes		
include		
defs_ou_decls		
def_ou_decl		
declaration ?		
vars_ou_foncs		
vars_ou_foncs_suite		
autres_vars		
parametres		
autres_parametres		
parametre		
decl_ou_instruction		
pointeur ?		
instruction		
if_then_else		
else ?		
while		
affectation_ou_appel		
variable_ou_appel		
deferencement		
affectation		
condition		
op_test		
return		
valeur_retour		
bloc		
vars_decls		
vars_decl		
instructions		
expression		
autres_termes		
terme		
autres_facteurs		
facteur		
parenthese_ou_cast		
var_ou_appel		
arguments ?		
arguments		
autres_arguments		

First – La valeur de la fonction First pour chacun des non-terminaux de la grammaire de C⋆.

5 Calcul de la fonction Follow

Pour rappel, la fonction FOLLOW calcule les lexèmes qui peuvent "suivre" un non-terminal. Cette fonction est surtout utile pour les productions pouvant dériver le mot nul. Ses valeurs ne serviront d'ailleurs que pour ces productions dans le calcul de la table de prédiction LL(1). Considérons un non-terminal X et une règle permettant de dériver le mot nul, disons typiquement $X \to \varepsilon$. Les valeurs de FOLLOW(X) permettent de déterminer si en fonction du prochain lexème à consommer (provenant de l'analyseur lexical) si l'analyseur syntaxique doit simplement retourner de la fonction courante (la fonction $X()$), sans consommer aucun lexème. Ceci doit se produire pour les lexèmes qui appartiennent à FOLLOW(X).

Pour calculer les valeurs de la fonction FOLLOW associées à chacun des non-terminal (disons X) il faut considérer **l'ensemble des règles de la grammaire** dont les productions contiennent le non-terminal considéré. Pour chacune de ces productions, il faut considérer les valeurs de la fonction FIRST (calculée à l'étape précédente) de la partie de la production qui suit le non-terminal X. Par exemple, soit la règle $Y \to \alpha X \beta$, il faut considérer la valeur de FIRST(β). Dans le cas où $\beta = \varepsilon$, on considère alors la valeur de FOLLOW(Y). Remplissez la table 4.

Non terminal	Valeur de la fonction Follow	Explication
programme		
includes		
include		
defs_ou_decls		
def_ou_decl		
declaration ?		
vars_ou_foncs		
vars_ou_foncs_suite		
autres_vars		
parametres		
autres_parametres		
parametre		
decl_ou_instruction		
pointeur ?		
instruction		
if_then_else		
else ?		
while		
affectation_ou_appel		
variable_ou_appel		
deferencement		
affectation		
cote_droit		
condition		
op_test		
return		
valeur_retour		
bloc		
vars_decls		
vars_decl		
instructions		
expression		
autres_termes		
terme		
autres_facteurs		
facteur		
parenthese_ou_cast		
var_ou_appel		
arguments ?		
arguments		
autres_arguments		

Figure 4 – La valeur de la fonction Follow pour chacun des non-terminaux de la grammaire de C⋆.

6 Calcul de la table de prédiction de l'analyseur LL(1) et implémentation de l'analyseur récursif descendant

6.1 Calcul de la table

Vous arrivez à la dernière étape qui consiste à calculer la table de prédiction de l'analyseur LL(1). L'algorithme à suivre est le suivant. Vous devez considérer les règles de la grammaire du langage, les unes à la suite des autres. Pour une règle donnée, disons $X \to \omega$ située en position i dans la grammaire, vous devez remplir des informations à la ligne X de la table :

1. Pour chaque lexème figurant dans First(X), indiquez l'entier i à l'intersection de la ligne X et de la colonne correspondant au lexème considéré.
2. Si (et seulement si), Null(ω) est vrai, alors pour chacun des lexèmes figurant dans Follow(X), indiquez à l'intersection de la ligne X et de la colonne correspondant au lexème considéré l'entier i. Utilisez une couleur différente pour ce type d'information par rapport aux précédentes (afin de vous souvenir que ces informations proviennent du calcul de Follow).

Remplissez la table 5. Lorsque vous l'aurez complètement remplie, assurez-vous qu'à chaque cellule de la table, il ne figure soit aucune, soit un unique numéro de règle (quelle que soit sa couleur). Si ce n'est pas le cas, votre grammaire est ambiguë. Il faut alors lever cette ambiguïté. Celle-ci peut provenir de :

— de la manière dont vous avez rédiger votre grammaire (elle n'est pas LL(1)).

— d'une ambiguïté dans le langage lui-même.

Que constatez-vous ? Que pensez-vous du programme suivant ?

```
void main(){
  int a,b ;

  a = _; // à remplir
  b= _; // à remplir
  if (a)
    if (b)
      printf("a et b\n");
    else
      print("(a et non b) ou bien non a ?\n");

}
```

6.2 Implémentation de l'analyseur syntaxique

Cette étape sera traitée lors de la prochaine séance de TP.

	Lexèmes																															
Non terminal	SYM_EOF	SYM_INCLUDE	SYM_EXTERN	SYM_IDENTIFIER	SYM_VOID	SYM_UINT64	SYM_SEMICOLON	SYM_COMMA	SYM_IF	SYM_ELSE	SYM_WHILE	SYM_RETURN	SYM_ASSIGN	SYM_LBRACE	SYM_RBRACE	SYM_PLUS	SYM_MINUS	SYM_ASTERISK	SYM_DIV	SYM_MOD	SYM_LPARENTHESIS	SYM_RPARENTHESIS	SYM_EQUALITY	SYM_LT	SYM_LEQ	SYM_GT	SYM_GEQ	SYM_NOTEQ	SYM_INTEGER	SYM_CHAR	SYM_STRING
programme																																
includes																																
include																																
defs_ou_decls																																
def_ou_decl																																
declaration ?																																
vars_ou_foncs																																
vars_ou_foncs_suite																																
autres_vars																																
parametres																																
autres_parametres																																
parametre																																
decl_ou_instruction																																
pointeur ?																																
instruction																																
if_then_else																																
else ?																																
while																																
affectation_ou_appel																																
variable_ou_appel																																
dereferencement																																
affectation																																
cote_droit																																
condition																																
op_test																																
return																																
valeur_retour																																
bloc																																
vars_decls																																
vars_decl																																
instructions																																
expression																																
autres_termes																																
terme																																
autres_facteurs																																
facteur																																
parenthese_ou_cast																																
var_ou_appel																																
arguments ?																																
arguments																																
autres_arguments																																

FIGURE 5 – La table de prédiction de l'analyseur LL(1) pour la grammaire du langage C⋆.