Compilation, langages et grammaires

L2 Informatique - UFR S.A.T

Pr. Ousmane THIARE

ousmane.thiare@ugb.edu.sn
http://www.ousmanethiare.com

16 avril 2020

Compilation, langages et grammaires

Chapitre XI : Compilation, langages et grammaires

Compilation, langages et grammaires
Le problème posé est...

Donner à un ordinateur un fichier contenant du texte, le lui faire lire et comprendre de manière à lui faire exécuter un certain nombre de tâches associées à ce fichier
$\Rightarrow$ On fait une compilation.

Détaillons succinctement les différentes phases d'une compilation...

L'analyse lexicale

On analyse le flux d'entrée de manière à le découper en unités lexicales, ou lexèmes.

Exemple. Dans if (temps=beau) etc., les unités lexicales sont « if », « (», « temps », « = », «beau», «)».

L'analyse syntaxique

Les contraintes à respecter pour que le texte soit compréhensible sont-elles respectées ? En d'autres termes, le flux de lexèmes est-il conforme à la syntaxe du langage utilisé (par comparaison à la grammaire du langage, c.f. ci-dessous) ?

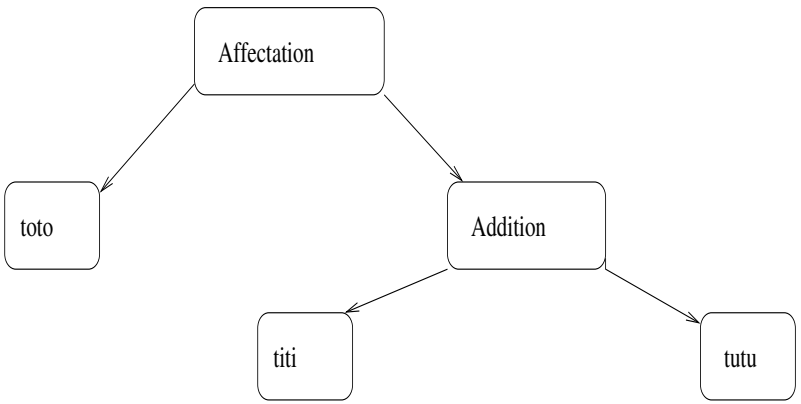
L'analyse sémantique

Reconnaître la signification d'un texte syntaxiquement correct : essayer de comprendre ce que cela signifie (le sens).

Cela implique notamment la transformation de la source en une forme utilisable, qui fasse apparaître le sens du texte.

Exemple. toto=titi+tutu ; est une instruction d'affectation à la variable « toto » d'une valeur exprimée par une expression algébrique, constituée de la somme des variables « titi » et « tutu ».

Remarque. Certains compilateurs utilisent des structures arborescentes :



Compilation proprement dite
Utiliser effectivement le résultat de l'analyse sémantique pour obtenir le résultat escompté, ce qui est demandé : production de code machine, traduction d'un texte dans une autre langue, etc.
Remarque. En général, ces différentes phases sont menées en parallèle.

Définition

Une grammaire est un ensemble de règles de syntaxe qui décrivent quels sont les constructions correctes qui sont possibles dans le langage utilisé, à l'aide de l'alphabet utilisé (alphabet ou vocabulaire).

Il existe de nombreux types de grammaires, et encore bien plus de formalismes exprimés pour repre ?senter cette grammaire.
Nous utiliseront pour commencer un seul symbolisme pour remprésenter les grammaires : la formalisation BNF (Backus-Naur Form).

Dans la syntaxe BNF, une grammaire est constituée d'un ensemble de règles.
Chaque règle est constituée :

- d'un premier membre,
- suivi du symbole de réécriture (: :=),
- suivi d'un second membre, qui peut être vide.

On utilise (et on distingue) des symboles terminaux et des symboles non-terminaux (ST et SNT).

Définition

Un symbole non terminal (SNT) est un symbole introduit (par commodité, ou plutôt par nécessité) par le rédacteur de la grammaire pour décrire les parties du fichier d'entrée qui représentent un tout logique et permettant de simplifier l'écriture de la grammaire.

Notation : Les symboles non-terminaux sont entourés par des chevrons : < >.
Le premier membre d'une règle de grammaire est un SNT (la règle en constitue la définition), le second membre est une famille ordonnée (éventuellement vide) de symboles, terminaux ou non.

Ainsi, chaque règle de la grammaire consiste en la définition d'un symbole non-terminal. Cette dernière est terminée quand tous les SNT ont reçu une définition. Une règle s'écrit finalement sous la forme :
<SNT> : := suite (éventuellement vide) de ST et SNT
Exemple. Voici un bout de grammaire (pour la définition d'une fonction) :
<fct> : := < type >< nom >" ("< parametres >")" < bloc >
<type> : := "int"
: := "char"

Définition

Parmi tous les SNT, l'un d'entre eux doit désigner l'ensemble du texte à analyser, on l'appelle axiome de la grammaire.

Exemple.

< programme en C > : :=< entete >< suite de fct >

La grammaire est terminée quand tous les SNT ont reçu au moins une définition.

« Les expressions correctes sont constituées d'un nombre quelconque, mais non nul, de 0, suivi d'un nombre quelconque, mais non nul, de 1. »

On conseille de suivre cette démarche :

1. Commencer par écrire la grammaire du langage (des expressions correctes).
2. Ecrire l'analyseur syntaxique pur.
3. Passer à l'analyseur syntaxique avec messages d'erreur.
4. Puis à l'analyseur syntaxique avec interprétation sémantique.

Exercice. Suivez ce cheminement :

1. Ecrivez cette grammaire.
2. Programmez, en C, l'analyseur syntaxique pur.
3. Ecrire le programme principal associé (le main).
4. Le modifier en analyseur syntaxique avec messages d'erreur, et adaptez le programme principal en conséquence.
5. Améliorez le programme pour qu'il devienne un analyseur syntaxique avec interprétation sémantique : ici, comptez le nombre de 0 et de 1, en cas de réussite.

$$
\begin{array}{rcl}
< \text{expression} > & ::= & < \text{groupe0} > < \text{groupe1} > \\
< \text{groupe0} > & ::= & \text{"0"} < \text{suite0} > \\
< \text{suite0} > & ::= & < \text{groupe0} > \\
 & ::= & \\
< \text{groupe1} > & ::= & \text{"1"} < \text{suite1} > \\
< \text{suite1} > & ::= & < \text{groupe1} > \\
 & ::= &
\end{array}
$$

Il faut :

- Subdiviser au maximum les expressions en sous-expressions cohérentes, en n'hésitant pas à multiplier les niveaux.
- Retarder au maximum les alternatives (en multipliant les niveaux) pour ne les faire intervenir que lorsqu'on ne peut plus faire autrement.

Voici le code de l'analyseur pur : il répond par « bon » ou « mauvais ».

```
#include <stdio.h>
char s[512] ;
char **ss ;
int expression (){
        if ( groupe0 ()==1)
            return groupe1 () ;
        return 0 ;
}
int groupe0 (){
        if (*ss == '0'){
            s ++ ;
            return suite0 () ;
        }
        return 0 ;
```

```
}
int suite0 (){
        if (groupe0() == 0)
            return 1 ;
        return 1 ;
}
```

Une fonction prévue pour analyser une sous-expression ne connaît pas ce qui précède et ne s'occupe pas de ce qui suit.

Passons au programme principal :

```
int main(){          printf ("Une expression a analyser ?
\n") ;
        scanf ("%s", s) ;
        ss=s ;
        if ( expression ()==1)
            if (*ss == '\0')
                printf ("Bon \n") ;
            else
                printf ("Mauvais \n") ;
}
```

Remarque. Toujours commencer par l'analyseur syntaxique pur.

```
int groupe0 (){
        if (*ss == '0'){
            ss ++ ;
            return suite0 () ;
}
printf ("L' expression doit commencer par 0 \n")
return 0 ;
int suite0 (){
        if (*ss == '0'){
            ss ++ ;
            return suite0 () ;
        }
        return 1 ;
}
```

Le programme principal devient alors :

```
int main(){          printf ("Une expression a analyser ? \
n") ;
        s c a n f ("% s" , s ) ;
        ss=s ;
        if (expression()==1)
            if (*ss == '\0')
                printf ("Bon \n") ;
            else if (*ss == '0')
                printf("Pas de 0 apres le(s) un(s). \n") ;
            else
                printf("Caractere interdit : % c \n",*ss) ;
}
```

```
int groupe (){
        if (*ss == '0'){
            ss ++ ;
            return 1+suite0 () ;
        }
        printf ("L' expression doit commencer par 0 \n")
        return 0 ;
}
int suite0(){
        if (*ss == '0'){
            ss++ ;
            return 1+suite0() ;
        }
        return 0 ;
}
```

Pareil pour groupe1 et suite1. Le programme principal devient alors :

```
int main(){
        printf ("Une expression à analyser ? \n")
        scanf("%s", s) ;
        ss=s ;
        Expression=expression() ;
        if (*ss=='\0')
            printf("Nombre de 0 : %d, nombre de 1 :
%d",expression.zero, expression.un) ;
        else
            printf("Caractere interdit : %c \n", *ss) ;
}
```

où la structure Expression et la fonction expression() sont ainsi définis :

```
struct Expression {
        int zero ;
        int un ;
}
struct Expression expression (){
        struct Expression a ;
        a.zero = groupe0() ;
        if (a.zero !=0)
            a.un = groupe1() ;
        return a ;
}
```