Dérécursivation

PCL - Travail de la grammaire

1 Algorithme de suppresion de la récursivité

1.1 Récursivité directe

Rappel 1.1 : Récursivité direct à gauche

Une grammaire est **immédiatement récursive à gauche** s'il existe un non terminal A et une règle de la forme $A \longrightarrow A\alpha$ où α est une chaîne de terminaux ou non terminaux quelconques.

Pour supprimer la récursivité à gauche :
On remplace les règles de la forme :

$$A \longrightarrow A\alpha_1 | \dots | A\alpha_n | \beta_1 | \dots | \beta_p$$

par les règles suivantes :

$$A \longrightarrow \beta_1 A' | \dots | \beta_p A'$$
$$A' \longrightarrow \alpha_1 A' | \dots | \alpha_n A' | \epsilon$$

où A' est un nouveau non terminal

1.2 Récursivité indirecte

Rappel 1.2 : Récursivité indirecte à gacuhe

Une grammaire est récursive à gauche s'il existe un non terminal A et une dérivation de la forme $A \xrightarrow{+} A\alpha$

Pour une grammaire sans règle $A \to \epsilon$ et **sans cycle**, on peut supprimer la récursivité gauche en appliquant l'algorithme suivant :

Ordonner les non terminaux de la grammaire $A_1 \dots A_n$
Pour $i = 1$ à n faire
  Pour $j = 1$ à $i - 1$ faire
    remplacer chaque règle de la forme $A_i \to A_j\alpha$ où $A_j \to \beta_1 | \dots | \beta_p$
    par $A_i \to \beta_1\alpha | \dots | \beta_p\alpha$
  fpour
  éliminer les récursivités à gauche immédiates des règles dont les membres gauches sont A_i
fpour

2 Dérécursiviation de la grammaire du `canAda`

On numérote les non terminaux :

- $A_1 =< fichier >$
- $A_2 =< decl >$
- $A_3 =< champs >$
- $A_4 =< type >$
- $A_5 =< params >$
- $A_6 =< param >$
- $A_7 =< mode >$

- $A_8 =< expr >$
- $A_9 =< instr >$
- $A_{10} =< operateur >$
- $A_{11} =< acces >$

Jusqu'à $A_7 =< mode >$, on ne tombe pas sur des règles qui entraine leur modification. Pour $A_8 =< expr >$, la règle $< expr > \rightarrow < expr >< operateur >< expr >$ est récursive gauche direct.

Remarque : La règle $< expre > \rightarrow < access >$ n'entraine pas de nouvelle règle car $A_8 =< expr >$ et $A_{11} =< acces >$ et que $8 < 11$.

On supprime la récursivité directe en introduisant $< expr1 >$

```
<expr> ::= <entier><expr1> | <caractère><expr1> true<expr1> | false<expr1> |
           null<expr1> | (<expr>)<expr1> | <accès><expr1>
           not<expr><expr1> | -<expr><expr1> |
           new<ident><expr1> |
           <ident>(<expr>,+)<expr1> |
           character ' val (<expr>)<expr1>

<expr1> ::= <operateur><expr><expr1> | ϵ
```

La prochaine règle qui va entrainer des modifications est $< acces > \rightarrow < expr > . < ident >$

Tout d'abord, il faut appliquer la modification donné par l'algorithme de dérécursivation indirecte :

```
<accès> ::= <entier><expr1>.<ident> |
            <caractère><expr1>.<ident> |
            true<expr1>.<ident> |
            false<expr1>.<ident> |
            null<expr1>.<ident> |
            (<expr>)<expr1>.<ident> |
            <accès><expr1>.<ident> |
            not<expr><expr1>.<ident> |
            -<expr><expr1>.<ident> |
            new<ident><expr1>.<ident> |
            <ident>(<expr>,+)<expr1>.<ident> |
            character ' val (<expr>)<expr1>.<ident> |
            <ident>
```

Cette modification entraîne l'apparition d'une règle récursive `<accès> ::= <accès><expr1>.<ident>`

On supprime donc la récursivité gauche directe en introduisant `<accès1>` :

```
<accès> ::= <entier><expr1>.<ident><accès1> |
            <caractère><expr1>.<ident><accès1> |
            true<expr1>.<ident><accès1> |
            false<expr1>.<ident><accès1> |
            null<expr1>.<ident><accès1> |
            (<expr>)<expr1>.<ident><accès1> |
            not<expr><expr1>.<ident><accès1> |
            -<expr><expr1>.<ident><accès1> |
            new<ident><expr1>.<ident><accès1> |
            <ident>(<expr>,+)<expr1>.<ident><accès1> |
            character ' val (<expr>)<expr1>.<ident><accès1> |
            <ident><accès1>

<accès1> ::= <expr1>.<ident>.<accès1> | ϵ
```

On arrive à la fin de l'algorithme. On a alors supprimé la récursivité gauche directe et indirecte en introduisant 2 nouveaux non terminaux `<expr1>` et `<accès1>`

3 Grammaire du canAda dérécursivée

```
<fichier>    ::= with Ada.Text_IO; use Ada.Text_IO;
                 procedure <ident> is <decl>*
                 begin <instr>+ end <ident>? ; EOF
<decl>       ::= type <ident> ;
               | type <ident> is access <ident> ;
               | type <ident> is record <champs>+ end record;
               | <ident>,+ : <type>(:= <expr>)? ;
               | procedure <ident> <params> ? is <decl>*
                 begin <instr>+ end <ident>?;
               | function <ident> <params>? return <type> is <decl>*
                 begin <instr>+ end <ident>?;
<champs>     ::= <ident>,+ : <type>;
<type>       ::= <ident>
<params>     ::= (<param>;+)
<param>      ::= <ident>,+ : <mode>? <type>
<mode>       ::= in | in out
<expr>       ::= <entier><expr1> | <caractère><expr1> true<expr1> | false<expr1> |
                 null<expr1> | (<expr>)<expr1> | <accès><expr1> |
                 not<expr><expr1> | -<expr><expr1> |
                 new<ident><expr1> |
                 <ident>(<expr>,+)<expr1> |
                 character ' val (<expr>)<expr1>
<expr1>      ::= <operateur><expr><expr1> | ^
<inst>       ::= <accès> := <expr> ;
               | <ident> ;
               | <ident> (<expr>,+);
               | return <expr>? ;
               | begin <instr>+ end;
               | if <expr> then <instr>+ (elsif <expr> then <instr>+)*
                 (else <instr>+)? end if;
               | for <ident> in reverse? <expr> .. <expr>
                 loop <instr>+ end loop;
               | while <expr> loop <instr>+ end loop;
<opérateur>  ::= = | /= | < | <= | > | >=
               | + | - | * | / | rem
<accès>      ::= <entier><expr1>.<ident><accès1> |
                 <caractère><expr1>.<ident><accès1> |
                 true<expr1>.<ident><accès1> |
                 false<expr1>.<ident><accès1> |
                 null<expr1>.<ident><accès1> |
                 (<expr>)<expr1>.<ident><accès1> |
                 not<expr><expr1>.<ident><accès1> |
                 -<expr><expr1>.<ident><accès1> |
                 new<ident><expr1>.<ident><accès1> |
                 <ident>(<expr>,+)<expr1>.<ident><accès1> |
                 character ' val (<expr>)<expr1>.<ident><accès1> |
                 <ident><accès1>
<accès1>     ::= <expr1>.<ident>.<accès1> | ^
```