Factorisation

PCL - Travail de la grammaire

1 Factorisation

Rappel 1.1 : Factorisation gauche

La **factorisation gauche** est la réécriture d'une gammaire en enlevant les règles qui commencent par le même token.

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

$$A \longrightarrow A\beta\delta \mid A\beta B \mid A\alpha$$
$$B \longrightarrow B\gamma \mid A\gamma$$

par les règles suivantes :

$$A \longrightarrow AA'$$
$$A' \longrightarrow \beta A'' \mid \alpha$$
$$A'' \longrightarrow \delta \mid B$$
$$B \longrightarrow B\gamma \mid A\gamma$$

où A' et A'' sont des nouveaux non terminaux.

2 Factorisation de la grammaire du canAda

On part de la grammaire précédemment dérécursivée

Les règles $< decl >$ $< mode >$ $< inst >$ $< operateur >$ et $< acces >$ nécessitent une/des factorisations.

On factorise $< decl >$ en introduisant $< decl1 >$ et $< decl2 >$ comme suit :

```
<decl>      ::= type <ident> <decl1>
              | <ident>,+ : <type>(:= <expr>)? ;
              | procedure <ident> <params> ? is <decl>*
                begin <instr>+ end <ident>?;
              | function <ident> <params>? return <type> is <decl>*
                begin <instr>+ end <ident>?;
<decl1>     ::= ;
              | is <decl2>
<decl2>     ::= access <ident> ;
              | record <champs>+ end record;
```

On factorise $< mode >$ en introduisant $< mode1 >$ comme suit :

```
<mode>      ::= in <mode1>
<mode1>     ::= out
              | ^
```

On factorise < *inst* > en introduisant < *inst*1 > comme suit :

```
<inst>       ::= <accès> := <expr> ;
               | <ident> <inst1>
               | 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;
<inst1>      ::= ;
               | (<expr>,+);
```

On factorise < *operateur* > en introduisant < *operateur*1 > comme suit :

```
<opérateur>  ::= =
               | / <opérateur1>
               | < <opérateur1>
               | > <opérateur1>
               | +
               | -
               | *
               | rem
<opérateur1> ::= =
               | ^
```

On factorise < *acces* > en introduisant < *acces*2 > comme suit :

```
<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> <accès2>
               | character ' val (<expr>)<expr1>.<ident><accès1>
<accès2>     ::= (<expr>,+)<expr1>.<ident><accès1>
               | <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> <decl1>
               | <ident>,+ : <type>(:= <expr>)? ;
               | procedure <ident> <params> ? is <decl>*
                 begin <instr>+ end <ident>?;
               | function <ident> <params>? return <type> is <decl>*
                 begin <instr>+ end <ident>?;
<decl1>      ::= ;
               | is <decl2>
```

```
<decl2>       ::= access <ident> ;
                | record <champs>+ end record;
<champs>      ::= <ident>,+ : <type>;
<type>        ::= <ident>
<params>      ::= (<param>;+)
<param>       ::= <ident>,+ : <mode>? <type>
<mode>        ::= in <mode1>
<mode1>       ::= 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> <inst1>
                | 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;
<inst1>       ::= ;
                | (<expr>,+);
<opérateur>   ::= =
                | / <opérateur1>
                | < <opérateur1>
                | > <opérateur1>
                | +
                | -
                | *
                | rem
<opérateur1>  ::= =
                | ^
<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> <accès2>
                | character ' val (<expr>)<expr1>.<ident><accès1>
<accès1>      ::= <expr1>.<ident>.<accès1> | ^
<accès2>      ::= (<expr>,+)<expr1>.<ident><accès1>
                | <accès1>
```