

# Factorisation

## PCL - Travail de la grammaire

### 1

### Factorisation

#### Rappel 1.1 : Factorisation gauche

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

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

$$\begin{aligned} A &\longrightarrow A\beta\delta \mid A\beta B \mid A\alpha \\ B &\longrightarrow B\gamma \mid A\gamma \end{aligned}$$

par les règles suivantes :

$$\begin{aligned} A &\longrightarrow AA' \\ A' &\longrightarrow \beta A'' \mid \alpha \\ A'' &\longrightarrow \delta \mid B \\ B &\longrightarrow B\gamma \mid A\gamma \end{aligned}$$

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

### 2

### Factorisation de la grammaire du canAda

On part de la grammaire précédemment factorisée

Les règles  $\langle decl \rangle$   $\langle mode \rangle$   $\langle inst \rangle$   $\langle operateur \rangle$  et  $\langle acces \rangle$  nécessitent une/des factorisations.

On factorise  $\langle decl \rangle$  en introduisant  $\langle decl1 \rangle$  et  $\langle decl2 \rangle$  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  $\langle mode \rangle$  en introduisant  $\langle mode1 \rangle$  comme suit :

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

On factorise  $\langle inst \rangle$  en introduisant  $\langle inst1 \rangle$  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  $\langle operateur \rangle$  en introduisant  $\langle operateur1 \rangle$  comme suit :

```

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

```

On factorise  $\langle acces \rangle$  en introduisant  $\langle acces2 \rangle$  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>

```

```

<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 <model>
<model>      ::= 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>      ::= <opérateur><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>

```