Le langage ZWEI

Types de données :

- deux type de base : *Int*, *Null*
- les classes.

Deux constructions de contrôle :

- ***while*** (<*cond*>) { ... }
- ***if*** (<*cond*>) <*expr*> [***else*** <*expr*>]

Opérateurs classiques :

- arithmétiques : $+$, $-$, $*$, $/$, $\%$,
- relationnels : $==$, $!=$, $<$, $\leq$, $>$, $\geq$,
- logiques : $!$, $\&\&$, $||$.

Exemples (1/2)

```
class Factorial {
    // Solution itérative
    Int factorial1(Int x) {
        Int p = 1;
        while (x > 0) {
            p = p * x;
            x = x - 1;
        }
        return p
    }
    // Solution récursive
    Int factorial2(Int x) { return
        if (x == 0) 1
        else x * this.factorial2(x - 1)
    }
}
```

Exemples (2/2)

```
class Example {
    Null main() {
        Factorial fac = new Factorial();
        Int x = 5;
        printInt(fac.factorial1(x));
        printInt(fac.factorial2(x));
    }
}
new Example().main()
```


Les classes (1/2)

Déclarer une classe :

```
class Rational {
    Int num;
    Int den;
    Rational add(Rational that) { return new Rational(
        this.num * that.den + that.num * this.den, this.den * that.den)
    }
}
```

Déclarer une sous-classe :

```
class ExtRational extends Rational {
    Rational mul(Rational that) { .. }
}
```

Les classes (2/2)

Créer un objet :

```
Rational r = new Rational(2, 3);
```

Sélectionner un champ :

```
Int x = r.num;
Int y = r.den;
```

Appeler une méthode :

```
Rational s = r.add(new Rational(1, 2));
```


Exemple de classe récursive : les listes (1/2)

Une liste représente un ensemble ordonnée de valeurs.

```
class List {
    Int isEmpty() { return this.isEmpty() }
    Int head() { return this.head() }
    List tail() { return this.tail() }
    List cons(Int x) { return this.cons(x) }
    // ..
}

class Cons extends List {
    Int head;
    Int tail;
    Int isEmpty() { return false }
    Int head() { return this.head }
    List tail() { return this.tail }
    List cons(Int x) { return new Cons(x, this) }
}
```

Exemple de classe récursive : les listes (2/2)

```
class Nil extends List {
    Int isEmpty() { return true }
    List cons(Int x) { return new Cons(x, this) }
}
```

Créations de listes :

```
List nil = new Nil();
List xs  = new Cons(2, new Cons(1, nil));
List ys  = nil.cons(1).cons(2);
...
```

Accès aux valeurs d'une liste *xs* :

```
xs.head()
xs.tail().head()
xs.tail().tail().head()
...
```


Syntaxe lexicale

```
input         = { inputelement }
inputelement  = whitespace
              | comment
              | token

token         = ident
              | number
              | ...     \{/* à compléter */\}
comment       = "/" "/" { cchar }
ident         = letter { letter | digit | "_" }
number        = "0" | digit1 { digit }
string        = "\"" { schar } "\""

whitespace    = " " | "\t" | "\f" | "\n"
letter        = "a" | ... | "z" | "A" | ... | "Z"
digit         = "0" | digit1
digit1        = "1" | ... | "9"
cchar         = \{tout caractère excepté "\n"\}.
schar         = \{tout caractère excepté "\n" et "\""\}.
```

Syntaxe (1/4)

Program	=	{ *ClassDecl* } *Expression*
ClassDecl	=	"***class***" *ident* ["***extends***" *ident*] "{" { *Member* } "}"
Member	=	*FieldDecl*
	|	*MethodDef*
FieldDecl	=	*Formal* ";"
MethodDef	=	*Formal* "(" *Formals* ")" *Block*
Formal	=	*Type ident*
Formals	=	[*Formal* { "," *Formal* }]
Block	=	"{" { *Statements* } ["***return***" *Expression*] "}"
Type	=	"*Int*"
	|	"*Null*"
	|	*ident*

Syntaxe (2/4)

Statement	=	"***while***" "(" *Expression* ")" "{" { *Statement* } "}"
	|	*Formal* "=" *Expression* ";"
	|	*ident* "=" *Expression* ";"
	|	*Expression* ";"
	|	"***printInt***" "(" *Expression* ")" ";"
	|	"***printChar***" "(" *Expression* ")" ";"
Expression	=	"***if***" "(" *Expression* ")" *Expression* ["***else***" *Expression*]
	|	*CmpExpression*
CmpExpression	=	[*SumExpression CompOp*] *SumExpression*
CompOp	=	"==" | "!=" | "<" | ">" | "≤" | "≥"

Syntaxe (3/4)

SumExpression = *Term*
| *SumExpression SumOp Term*

SumOp = "+" | "−" | "||"

Term = [*Term ProdOp*] [*NegateOp*] *Factor*

NegateOp = "−" | "!"

ProdOp = "∗" | "/" | "%" | "&&"

Syntaxe (4/4)

Factor = *ident*
| *number*
| *string*
| "**true**"
| "**false**"
| "**this**"
| "**null**"
| "**readInt**"
| "**readChar**"
| "(" *Expression* ")"
| *Block*
| "**new**" *ident Params*
| *Factor* "." *ident Params*
| *Factor* "." *ident*

Params = "(" *Expressions* ")"

Expressions = [*Expression* { "," *Expression* }]