Recursive Descent Expression Parser

Jeg tok utgangspunkt i Wikipedia's artikkel og snekret sammen en parser på høsten ifjor. Den bygget jeg ut etterhvert som jeg fikk behov. Til slutt var det en innfløkt masse kode.

Jeg mente å ha hørt at man kunne parse expression ved å tildele operatorene 'precedence level'. Så det måtte jeg prøve.

Koden min reduserte jeg ved en slags 'pattern matching' – ved at jeg gjenkjente et mønster og gjorde den enkel og rekursiv.

Jeg prøvde meg fram med å tildele 'precedence level' til operatorene. Mye feiling.

Siste runde var i dialog med Stein. Han stilte ubehaglige spørsmål – jeg rettet koden

Slik står det idag:

Expression Parsing med Operator Precedence Levels.

Jeg endte med denne tabelen:

```
14      := :-
13      OR_ELSE
12      AND_THEN
11      EQV
10      IMP
 9      OR
 8      AND
 7      NOT
 6      < <= > >= = <> == =/= IS IN
 5      + -
 4      * / //
 3      **
 2      QUA . Remote access
```

Og denne algoritmen:

```
private static Expression parseBinaryOperation(final int level) {

  Expression expr=(level>0)?parseBinaryOperation(level-1):parsePrimaryExpression();

  while(Parser.acceptBinaryOperator(level))
  { KeyWord opr=Parser.prevToken.getKeyWord();
    if(level==0) expr=new BinaryOperation(expr,opr,parsePrimaryExpression());
    else if(isRightAssossiativ(opr))
          expr=new BinaryOperation(expr,opr,parseBinaryOperation(level));
    else expr=new BinaryOperation(expr,opr,parseBinaryOperation(level-1));
  }
  return(expr);
}
```

Denne ble til etter intens quiz fra Stein Krogdahl som jeg måtte parere.

Jeg vet fortsatt ikke hvorfor den virker, men alt tyder på at det gjør den !

Forklaring til algoritmen på forrige side.

- Jeg bruker en Scanner som leverer Tokens som holder KeyWord og en Verdi.

- Alle Operatorer leveres som TOKEN med KeyWord som (PLUS,MINUS, ... MULT, ... DOT, QUA, ...) og en verdi svarende til Precedence Level

- Jeg bruker en Parser ala' Wirth med to Primitiver:
  Accept(KeyWord) og Expect(KeyWord)

  Se nærmere her: https://en.wikipedia.org/wiki/Recursive_descent_parser

- Jeg har laget en ny Funksjon Parser.acceptBinaryOperator(level) som er definert slik:

```
public static boolean acceptBinaryOperator(int level) {
      Object val = Parser.currentToken.getValue();
      if (val != null && val instanceof Precedence) {
            int operatorLevel = ((Precedence) val).getValue();
            if (operatorLevel == level) {
                  nextSymb();
                  return (true);
            }
      }
      return (false);
}
```

  Altså den 'aksepterer (og spiser)' et Token hvis det er en operator med oppgitt precedence level.

- Expression-treet har noder som BinaryOperation(lhs,opr,rhs), UnaryOperation(opr,rhs) (... og en del andre for å holde diverse Primary Expressions).

  opr er TOKEN, lhs og rhs er Expression.

- Metoden `isRightAssossiativ er definert slik:`

```
private static boolean isRightAssossiativ(KeyWord opr {
     return(opr==KeyWord.ASSIGNVALUE
         || opr==KeyWord.ASSIGNREF
         || opr==KeyWord.EXP);
}
```

Til info: parsePrimaryExpression ser slik ut:

```
public static Expression parsePrimaryExpression()
{ // PrimaryExpression =  ( Expression ) | Constant | ObjectGenerator
  //                    | LocalObject | UnaryOperation | Variable | SubscriptedVariable
  //    Constant = IntegerConstant | RealConstant | CharacterConstant
  //             | TextConstant | BooleanConstant | SymbolicValue
  //    BooleanConstant = TRUE | FALSE
  //    Boolean-secondary =  [ NOT ]  Boolean-primary
  //    SymbolicValue = NONE | NOTEXT
  //    LocalObject = THIS ClassIdentifier

  if (Parser.accept(KeyWord.BEGPAR)) { Expression expr=parseExpression();
                                       Parser.expect(KeyWord.ENDPAR); return(expr); }
  else if(Parser.accept(KeyWord.INTEGERKONST)) return(new
                                  Constant(Type.Integer,Parser.prevToken.getValue()));
  else if(Parser.accept(KeyWord.REALKONST)) return(new
                                  Constant(Type.LongReal,Parser.prevToken.getValue()));
  else if(Parser.accept(KeyWord.BOOLEANKONST)) return(new
                                  Constant(Type.Boolean,Parser.prevToken.getValue()));
  else if(Parser.accept(KeyWord.CHARACTERKONST)) return(new
                                  Constant(Type.Character,Parser.prevToken.getValue()));
  else if(Parser.accept(KeyWord.TEXTKONST)) return(new
                                  Constant(Type.Text,Parser.prevToken.getValue()));
  else if(Parser.accept(KeyWord.NONE)) return(new Constant(Type.Ref,null));
  else if(Parser.accept(KeyWord.NOTEXT)) return(new Constant(Type.Text,null));
  else if(Parser.accept(KeyWord.NEW)) return(ObjectGenerator.parse()); // TODO
  else if(Parser.accept(KeyWord.THIS)) return(LocalObject.acceptThisIdentifier());
  else if(Parser.accept(KeyWord.PLUS)) return(parseUnaryOperation());
  else if(Parser.accept(KeyWord.MINUS)) return(parseUnaryOperation());
  else if(Parser.accept(KeyWord.NOT))
       { // Boolean-secondary =  [ NOT ]  Boolean-primary
         Expression expr=parseBooleanPrimary();
         expr=new UnaryOperation(KeyWord.NOT,expr);
         return(expr);

       }
  else if(Parser.accept(KeyWord.IF))
    { Expression condition=parseExpression();
      Parser.expect(KeyWord.THEN); Expression thenExpression=parseSimpleExpression();
      Parser.expect(KeyWord.ELSE); Expression elseExpression=parseExpression();
      Expression expr=new
                ConditionalExpression(Type.Boolean,condition,thenExpression,elseExpression);
      if(Option.TRACE_PARSE) Util.TRACE("Expression: parsePrimaryExpression, result="+expr);
      return(expr);
    }
  else
    { String ident=acceptIdentifier();
      if(ident!=null) return(Variable.parse(ident));
      return(null);
    }
}
```

Og *parseBooleanPrimary*:

```
 private static Expression parseBooleanPrimary()
  { return(parseBooleanPrimary(7)); }

  private static Expression parseBooleanPrimary(final int level)
  { Expression expr=(level>0)?parseBooleanPrimary(level-1):parsePrimaryExpression();
    while(Parser.acceptOperatorLevel(level))
    { KeyWord opr=Parser.prevToken.getKeyWord();
      if(level==0) expr=new BinaryOperation(expr,opr,parsePrimaryExpression());
      else expr=new BinaryOperation(expr,opr,parseBooleanPrimary(level-1));
    }
    return(expr);
  }
```

Og *parseUnaryOperation*:

```
private static Expression parseUnaryOperation()
  { Expression expr=new
            UnaryOperation(Parser.prevToken.getKeyWord(),parseBinaryOperation(4));
    return(expr);
  }
```