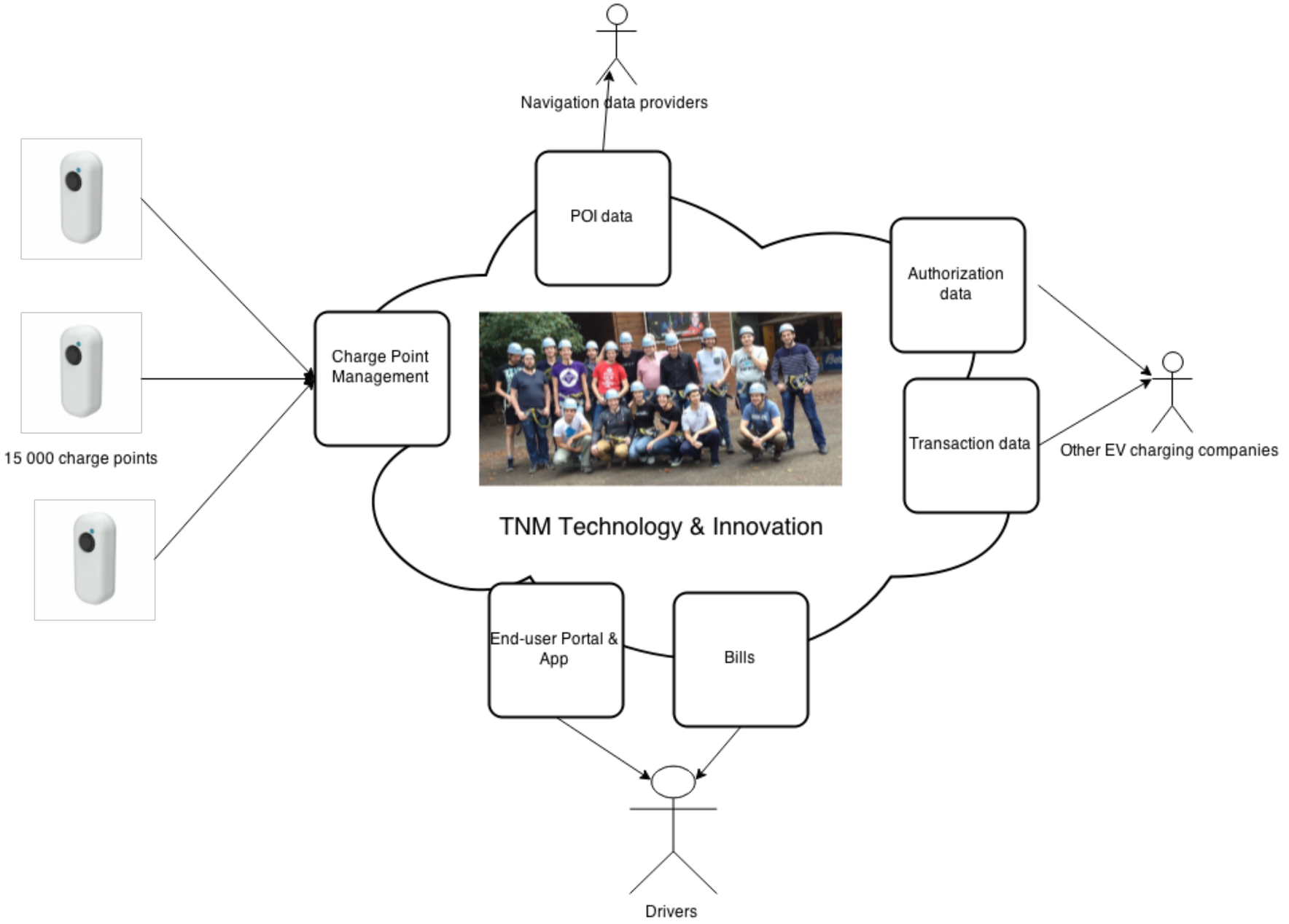
THENEWMOTION

laadoplossingen thuis, op het werk en onderweg


Navigation data providers
POI data
Authorization data
Charge Point Management
Transaction data
Other EV charging companies
15 000 charge points
TNM Technology & Innovation
End-user Portal & App
Bills
Drivers

Why Scala?

It's a language to get things done!

- TMTOWTDI
- Use Java libraries
- Creative hacker community

Settlement:

Tackling complexity with Scala

What is settlement

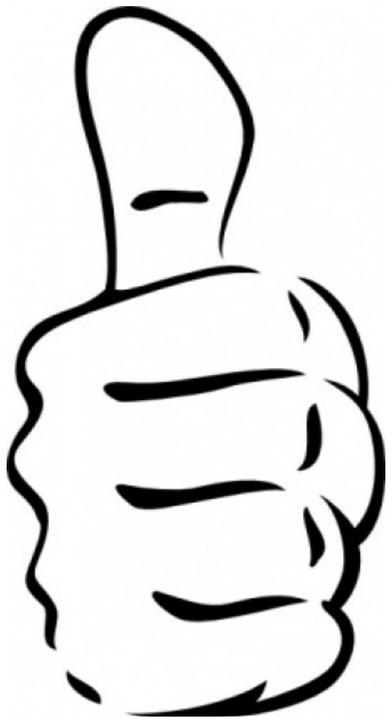
From monolitic PHP script to custom rule DSL

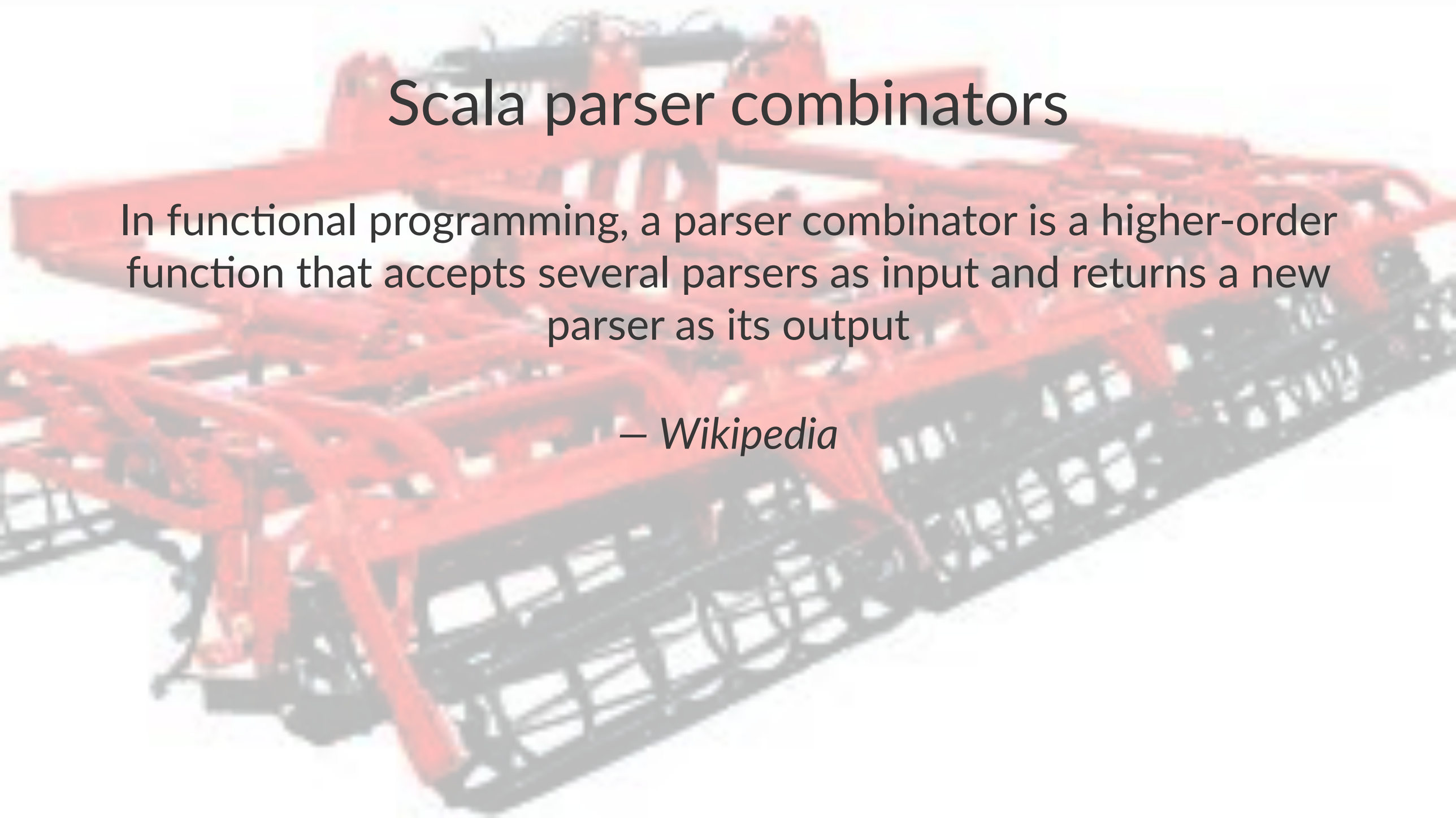
Rule templates

- *pays_amount = [algebraic expression] when [boolean condition]*
- *receives_amount = [algebraic expression] when [boolean condition]*

Rule sample

*receives_amount = bounded(0.5 + 0.2 * duration_minutes, 0, 20) when duration < 10 minutes and chargepoint_id in ["idA", "idB"]*





Scala parser combinators

In functional programming, a parser combinator is a higher-order function that accepts several parsers as input and returns a new parser as its output

— *Wikipedia*

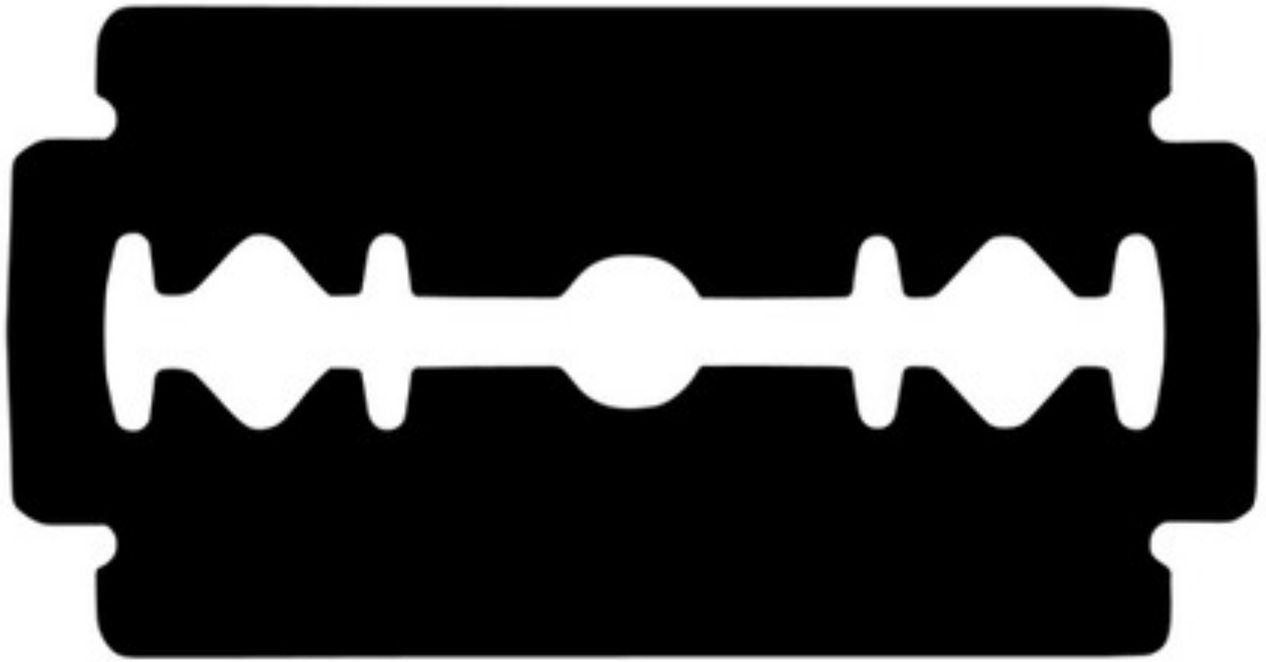
New token types

- Money tokens: 1 EUR, 2 cent
- Energy tokens: 1 kwh, 10 wh
- Duration tokens: 1 day, 3 hour, 10 second

Example of lexer

```
trait MoneyLexical extends BaseLexical with MoneyTokens{
  override def token: Parser[Token] = moneyToken | super.token

  def moneyToken = floatingToken ~ whitespace ~ (eur | cents) <~ endOfIdentifier ^^ {
    case amount ~ _ ~ f => f(amount.chars)
  }

  private def eur: Parser[String => Token] = string("EUR") ^^^ strToMoney(BigDecimal("1"), "EUR")

  private def cents: Parser[String => Token] = string("cent") ^^^ strToMoney(BigDecimal("100"), "cent")

  // Implementation omitted for brevity
  private def strToMoney(divider: BigDecimal, currencyUnit: String)(amount: String): Token = ???
}

trait MoneyTokens extends Tokens {
  case class MoneyToken(chars: String, money: Money) extends Token
}
```

DIVIDE ET



IMPERA

Dsls

- SimpleConditionDsl[T]
- ComposedConditionDsl[T]
- AlgebraDsl[T]
- AssignmentDsl[S, T]

ComposedConditionDsl[T]

```
trait ComposedConditionDsl[T] { this: MyStandardTokenParsers =>
  lexical.delimiters += ("(", ")")
  lexical.reserved += ("not", "and", "or")

  // 'in' pin
  protected def composable_boolean_condition: Parser[Condition[T]]

  // 'out' pin
  protected def bool_logic_spec: Parser[Condition[T]] = or_spec

  private def composable_term: Parser[Condition[T]] = parenthised_spec | composable_boolean_condition

  private def inverse_spec: Parser[Condition[T]] = "not" ~> composable_term ^^ {
    case x => !x
  }

  private def and_spec: Parser[Condition[T]] = rep1sep(inverse_spec | composable_term, "and") ^^ {
    case list => list.reduce(_.&&(_))
  }

  private def or_spec: Parser[Condition[T]] = rep1sep(and_spec, "or") ^^ {
    case list => list.reduce(_.||(_))
  }

  private def parenthised_spec: Parser[Condition[T]] = "(" ~> or_spec <~ ")"
}
```

Test implementation

```
object BooleanDsl extends MyStandardTokenParsers with ComposedConditionDsl[Any] {
  lexical.reserved +=("true", "false")

  def true_spec: Parser[Condition[Any]] = "true" ^^ {
    t => new Condition[Any] {
      override def apply(s: Any): Boolean = true
    }
  }

  def false_spec: Parser[Condition[Any]] = "false" ^^ {
    t => new Condition[Any] {
      override def apply(s: Any): Boolean = false
    }
  }

  protected def composable_boolean_condition: Parser[Condition[Any]] = true_spec | false_spec

  def entry: Parser[Condition[Any]] = bool_logic_spec

  def parse(rule: String): ParseResult[Condition[Any]] = {
    phrase(entry)(new lexical.Scanner(rule))
  }
}
```

Test samples

- true and false
- not false or true
- not (true or true)
- not((true or false and true) and false)

SimpleConditionDsl test samples

```
case class X(x: Duration,
             y: Option[Duration])
```

- x = 2 minute
- y > 1 hour
- x <= y

SimpleConditionDsl test samples

```
case class Y(x: Energy,
             y: Option[Energy])
```

- x = 2 kwh
- y > 10 wh
- x != y

AlgebraDsl test implementation

```
object TestConditionDsl extends AlgebraDsl[X] {
  lexical.reserved +=("x", "y", "min", "max")

  protected override def bigd_operands: Parser[X => BigDecimal] = ("x" | "y") ^^ {
    case "x" => _.x
    case "y" => _.y
  }

  def parse(rule: String): ParseResult[X => BigDecimal] = {
    phrase(algebra_spec)(new lexical.Scanner(rule))
  }
}
```

AlgebraDsl test samples

```
case class X (x: BigDecimal, y: BigDecimal)
```

- x + 10
- y * 5
- (x * y) / (x - y)
- min(x / y, x * y)

AssignmentDsl test implementation

```
object TestConditionDsl extends AssignmentDsl[Any, X] {
  lexical.reserved +=("x", "y")

  override protected def bigd_setters: Parser[(X, BigDecimal) => X] =
    ("x" ^^^ {(s: X, v: BigDecimal) => s.copy(x = v)}) |
    ("y" ^^^ {(s: X, v: BigDecimal) => s.copy(y = v)}))

  override protected def expression: Parser[Any => BigDecimal] = bigd_literals ^^ {
    case lit => _ => lit
  }

  def parse(rule: String): ParseResult[(Any, X) => X] = {
    phrase(multi_assign_spec)(new lexical.Scanner(rule))
  }
}
```

AssignmentDsl test samples

```
case class X(x: BigDecimal, y: BigDecimal)
```

- x = 1
- y = 2; x = 3
- [x, y] = 4

SettlementDsl

Finally settlement dsl parses rules into a form:

```
class TransformIf[S, T](transform: (S, T) => T, condition: S => Boolean) extends ((S, T) => Option[T]) {
  override def apply(s: S, t: T): Option[T] = if (condition(s)) Some(transform(s, t)) else None
}
```