2023-12-18 - DSL esterni, Parser combinators, Grammatica, Regole di produzione

DSL esterni

Parser combinators

Un parser è una funzione che accetta delle **stringhe in input** e restituisce una **struttura "parsata"** *(spesso un parse tree)*.

Un **parser combinator** è una funzione che prende in **input più parser** e restituisce un **nuovo parser**, ottenendo un *parser a discesa ricorsiva* (**recursive descent parsing strategy**).

Grammatica e Regole di produzione

È necessario definire una **grammatica** per il **DSL**, ovvero una **quadrupla** composta da:

- **simboli terminali** *(terminal symbols)*: le keyword, i simboli di punteggiatura e gli identificatori del linguaggio, che compongono la sintassi e permettono di capire al parser cosa apparirà nel codice
- **simboli non terminali** *(non-terminal symbols)*: simboli o placeholder per astrarre un gruppo di simboli terminali che rappresentano un costrutto del linguaggio *(espressioni, istruzioni, funzioni, ...)*
- **regole di produzione** *(production rules)*: la sintassi del linguaggio, definizione di come i simboli non terminali possono venir rimpiazzati *(e quindi parsati)* da simboli terminali o non terminali *(generando ricorsione)*. Sono coppie nella forma $A \to \alpha$, dove A è un simbolo non terminale e α è una sequenza di simboli terminali e non terminali
- **simbolo iniziale** o assioma di partenza *(start symbol)*: simbolo non terminale che rappresenta la radice della sintassi, permette di derivare l'intero programma applicando le regole di produzione. Funge da entry point per il parsing

Le **regole di produzione** (nel formato $A \to \alpha$) vengono definite:

- parte sinistra A:
  - **simboli non terminali**: astrazione di una parte di linguaggio, il *"nome"*
- parte destra α:
  - **simboli non terminali**: *"sotto-regole"* di cui la regola attuale è composta
  - **simboli terminali**: le keyword che permettono al parser di capire qualche regola applicare
  - **alternative** *(alternatives)*: diverse possibilità all'interno della stessa regola (`|`)
  - **sequenze** *(sequences)*: più simboli che appaiono nell'ordine stabilito dalla regola
  - **ripetizioni** *(repetitions)*: possibile ripetizione di sequenze (`{ , rule } | epsilon`)

Esempio: meccanismo di calcolo stipendio dipendenti

Dobbiamo "parsare" il seguente codice:

```
paycheck for employee "Buck Trends" is salary for 2 weeks minus deductions for {
  federal income tax              is 25. percent of gross,
  state income tax                is 5. percent of gross,
  insurance premiums              are 500. in gross currency,
  retirement fund contributions   are 10. percent of gross
}
```

Grammatica:

```
paycheck       = empl gross deduct
empl           = paycheck for employee employeeName
gross          = is salary for duration
deduct         = minus deductions for { deductItems }
employeeName   = " name name "
name           = ...
duration       = decimalNumber weeksDays
weeksDays      = week | weeks | day | days
deductItems    = deductItem { , deductItem } | epsilon
deductItem     = deductKind deductAmount
```

```
deductKind     = tax | insurance | retirement
tax            = fedState income tax
fedState       = federal | state
insurance      = insurance premiums
retirement     = retirement fund contributions
deductAmount   = percentage | amount
percentage     = toBe doubleNumber percent of gross
amount         = toBe doubleNumber in gross currency
toBe           = is | are
decimalNumber  = ...
doubleNumber   = ...
```

Parser combinators in Scala

Per scrivere dei parser combinators in Scala è possibile utilizzare la libreria `scala-parser-combinators`:

- le **regole di produzione** (quindi ogni simbolo non terminale) vengono definite come funzioni, che restituiscono un `Parser[T]`
- i **simboli terminali** come semplici stringhe
- le **alternative** con l'operatore `|`
- le **sequenze** attraverso gli operatori `~`:
  - `~` che mantiene sia i simboli a destra che a sinistra
  - `~>` che mantiene solo i simboli a destra (scartando quelli a sinistra)
  - `<~` che mantiene solo i simboli a sinistra (scartando quelli a destra)
- le **ripetizioni** attraverso le funzioni `rep()` (senza separatore) e `repsep()` (con separatore)
- i simboli **opzionali** attraverso la funzione `opt()`

Esempio: meccanismo di calcolo stipendio dipendenti

```
import scala.util.parsing.combinator._

// stringLiteral, decimalNumber, floatingPointNumber from JavaTokenParsers
class PayrollParserCombinatorsV1 extends JavaTokenParsers {
    def paycheck = empl ~ gross ~ deduct
    def empl = "paycheck" ~> "for" ~> "employee" ~> employeeName
    def gross = "is" ~> "salary" ~> "for" ~> duration
    def deduct = "minus" ~> "deductions" ~> "for" ~> "{" ~> deductItems <~ "}"
    def employeeName = stringLiteral
    def duration = decimalNumber ~ weeksDays
    def weeksDays = "weeks" | "week" | "days" | "day"
    def deductItems = repsep(deductItem, ",")
    def deductItem = deductKind ~> deductAmount
    def deductKind = tax | insurance | retirement
    def tax = fedState <~ "income" <~ "tax"
    def fedState = "federal" | "state"
    def insurance = "insurance" ~> "premiums"
    def retirement = "retirement" ~> "fund" ~> "contributions"
    def deductAmount = percentage | amount
    def percentage = toBe ~> doubleNumber <~ "percent" <~ "of" <~ "gross"
    def amount = toBe ~> doubleNumber <~ "in" <~ "gross" <~ "currency"
    def toBe = "is" | "are"
    def doubleNumber = floatingPointNumber
}
```

Utilizzando il parser definito attraverso la funzione `parseAll(start_symbol, input)`, verrà restituita un'istanza di:

- `Success[+T]` in caso il parsing vada a **buon fine**, che contiene i **dati** "parsati" e il resto dell'input non parsato *(normalmente vuoto)*
- `Failure` o `Error` in caso di **fallimento** del parsing

```
val p = new Parser

p.parseAll(start_symbol, input) match {
    case p.Success(parsed, rem) => ... // successo
    case x => ... // fallimento
}

// esempio:
val p = new PayrollParserCombinatorsV1

p.parseAll(p.paycheck, new FileReader("input.txt")) match {
    case p.Success(parsed, rem) => ... // successo
    case x => ... // fallimento
}
```

Ogni parser, dopo aver estratto i simboli necessari, può effettuare delle **operazioni**, attraverso l'**applicazione di funzioni** al risultato del parser, definite dopo l'**operatore** `^^`.

Il **tipo** restituito dal parser dipende dal tipo **restituito da queste funzioni**. In caso non venga definita nessuna funzione, il risultato è `Parser[String]`.

Esempio: meccanismo di calcolo stipendio dipendenti

```
class PayrollParserCombinators(val employees: Map[Name,Employee]) extends JavaTokenParsers {
    var currentEmployee: Employee = null
    var grossAmount: Money = Money(0)

    /** @return Parser[(Employee, Paycheck)] */
    def paycheck = empl ~ gross ~ deduct ^^ {
        case em ~ gr ~ de => (em, Paycheck(gr, gr-de, de))
    }

    /** @return Parser[Employee] */
    def empl = "paycheck" ~> "for" ~> "employee" ~> employeeName ^^ { name =>
        val names = name.substring(1, name.length-1).split(" ")
        val n = Name(names(0), names(1));
        if (!employees.contains(n))
          throw new UnknownEmployee(n)
        currentEmployee = employees(n);
        currentEmployee
    }

    /** @return Parser[Money] */
    def gross = "is" ~> "salary" ~> "for" ~> duration ^^ { dur =>
        grossAmount = salaryForDays(dur);
        grossAmount
    }

    def deduct = "minus" ~> "deductions" ~> "for" ~> "{" ~> deductItems <~ "}"

    /** "stringLiteral" provided by JavaTokenParsers
     * @return Parser[String] */
    def employeeName = stringLiteral

    /** "decimalNumber" provided by JavaTokenParsers
     * @return Parser[Int] */
    def duration = decimalNumber ~ weeksDays ^^ {
        case n ~ factor => n.toInt * factor
    }
```

```
    def weeksDays = weeks | days

    def weeks = "weeks?".r ^^ { _ => 5 }

    def days = "days?".r ^^ { _ => 1 }

    /** @return Parser[Money] */
    def deductItems = repsep(deductItem,",") ^^ { items =>
        items.foldLeft(Money(0)){_ + _}
    }

    def deductItem = deductKind ~> deductAmount

    def deductKind = tax | insurance | retirement

    def tax = fedState <~ "income" <~ "tax"

    def fedState = "federal" | "state"

    def insurance = "insurance" ~> "premiums"

    def retirement = "retirement" ~> "fund" ~> "contributions"

    def deductAmount = percentage | amount

    def percentage = toBe ~> doubleNumber <~ "percent" <~ "of" <~ "gross" ^^ { percentage =>
        grossAmount * Type2Money.double2Money(percentage / 100.0)
    }

    def amount = toBe ~> doubleNumber <~ "in" <~ "gross" <~ "currency" ^^ { Money(_) }

    def toBe = "is" | "are"

    def doubleNumber = floatingPointNumber ^^ { _.toDouble }

    def salaryForDays(days: Int) =
        (currentEmployee.annualGrossSalary / Type2Money.double2Money(260.0))
        * Type2Money.int2Money(days)
}
```

Esempio completo: Exercises/Various/Paycheck.scala