Algorithm Unibo

Le monadi (secondo me) e il parsing di un linguaggio di programmazione

Luca Borghi

Università di Bologna

t.me/algo_unibo

Parsing

Parsing di un linguaggio

Un algoritmo di parsing legge e trasforma una sorgente (solitamente sotto forma di stringa) in una struttura dati che funge da rappresentazione astratta della sorgente. In particolare, il parsing di un linguaggio di programmazione consiste nella seguente funzione:

$$parser : stringa -> ast$$

Lex e Yacc

Esiste un modo classico per effettuare il parsing di un linguaggio, in cui vengono utilizzati due programmi specifici:

- **Lex**, un programma che prende in input la lista di token del linguaggio e genera un analizzatore lessicale, ovvero un altro programma in grado di riconoscere i singoli costrutti di un programma
- **Yacc**, un programma che prende in input la grammatica del linguaggio e genera un parser

Dato che è necessario definire una lista di espressioni regolari (per i token) e una grammatica, tale approccio è completamente dichiarativo

Parser combinators

Un *parser combinator* è una funzione che si occupa di effettuare il parsing di un pezzo di sorgente, consumandolo. Quest'approccio prevede di comporre dei parser combinator che si occupano di fare il parsing di elementi *atomici*, creando man mano funzioni in grado di parsare elementi sempre più complessi

Parsec

Noi utilizzeremo Parsec, una libreria scritta in Haskell che si basa sul concetto di parser combinator monadico:
https://hackage.haskell.org/package/parsec

ParsecT: la monade principale

Parsec espone una monade, la quale rappresenta un parser combinator:

```
data ParsecT s u m a
```

dove `s` è il tipo del sorgente (di solito è `String`), `u` è il tipo dello stato che l'utente può utilizzare mentre viene effettuato il parsing (se non c'è necessità di utilizzare uno stato interno, basta istanziare la variabile di tipo con `()`), `m` è il sotto-contesto in cui viene inglobato il tipo di ritorno `a`. È necessario tenere conto che, di solito, i parser combinator **consumano** l'input

ParseError: error handling

Se un parser combinator fallisce, non dobbiamo preoccuparci di gestire l'errore, sarà Parsec stesso a gestirlo. Infatti, per eseguire effettivamente un parser combinator, utilizziamo:

```
runParserT
    :: Stream s m t
    => ParsecT s u m a
    -> u
    -> SourceName
    -> s
    -> m (Either ParseError a)
```

il quale ritorna già un valore di errore (`ParseError`) se il parser combinator fallisce. Perciò, nella maggior parte dei casi, si può utilizzare direttamente il tipo:

```
type Parsec s u = ParsecT s u Identity
```

Esempio 1

Parsing di un sorgente csv con stringhe contenenti solo le lettere 'A', 'C', 'G', 'T':

```
dnaBase :: Parsec String () Char
dnaBase =
    choice [ try $ char 'A', try $ char 'C', try $ char 'G', char 'T' ]

csv :: Parsec String () [String]
csv = (many1 dnaBase) `sepBy` (symbol ",")
```

Esempio 2 (1)

Parsing di un blocco html con tag parametrici:

```
start :: Parsec String () t -> Parsec String () t
start tag = do
    char '<'
    res <- tag
    char '>'

end :: Parsec String () t -> Parsec String () t
end tag = do
    string "<\\"
    res <- tag
    char '>'
```

Esempio 2 (2)

```
block
    :: Parsec String () t
    -> Parsec String () a
    -> Parsec String () (t, a, t)
block tag inner = do
    s <- start tag
    skipMany space
    i <- inner
    skipMany space
    e <- end tag
    return (s, i, e)
```

Esercizio 6

Proviamo a fare un piccolo interprete che prenda in input operatori operazioni aritmetiche e li esegua
Esempio di BNF (Backus Naur Form) per sta grammatica:

```
Arith := Int | Int + Int | Int - Int | Int * Int | Int / Int
```

Esempio

Se eseguo 'Main testo.txt' con 6+5 dentro a test.txt, l'interprete deve essere in grado di tornare 11.
Extra: fare in modo che capisca anche cose come 6+1+3+1 **(Difficile) bisogna conoscere Parsec!**

Demo

Esercizio 6.5

Definire una grammatica di un linguaggio, definite la struttura dati per l'AST e costruire il parser. Se volete, di seguito c'è una grammatica già definita da cui prendere spunto e che potete estendere come volete. Il simbolo `Int` rappresenta un qualsiasi numero intero, il simbolo `Char` rappresenta un qualsiasi carattere, `Sym` è per gli identificatori delle variabili, `Con` è per gli identificatori dei costruttori per un tipo, `Type` è per gli identificatori dei tipi

Grammatica esercizio 6

```
Decl := fun Sym(Sym : Type, ..., Sym : Type) -> Type { Expr }
      | type Type = Con(Type, ..., Type), ..., Con(Type, ..., Type)

Expr := let Sym = Expr | Expr ; Expr | Sym(Expr, ..., Expr)
      | Con(Expr, ..., Expr) | Sym | Con | Arith | Char | (Expr)

Arith := Int | Int + Int | Int - Int | Int * Int | Int / Int
```

Bibliografia

- A.Asperti, G.Longo. Category Theory for the Working Computer Scientist. M.I.T.Press. 1991: https://www.cs.unibo.it/~asperti/PAPERS/book.pdf
- Download Haskell toolchain: https://www.haskell.org/ghcup/install/
- Haskell Hierarchical Libraries: https://downloads.haskell.org/ghc/latest/docs/libraries/
- Parsec library: https://hackage.haskell.org/package/parsec