Instrumente dedicate învățării asistate de calculator a limbilor străine bazate pe combinatorii de parsere

Dan Popa

Univ. "Vasile Alecsandri", Calea Mărăşeşti 157, Bacău, Romania
danvpopa@ub.ro, popavdan@yahoo.com

Rezumat : Un set de combinatori de parsere implementat in limbajul funcțional de înalt nivel Haskell se pot folosi pentru implementarea de exerciții din domeniul lingvisticii, întrucât combinatorii de parsere permit crearea de verificatoare de sintaxă și de interpretoare modulare.

Introducere

Principalele etape ale proiectului sunt următoarele: Pasul I: Prezentarea unei metodologii (care să fie de asemenea utilizabilă de către lingviști) pentru transformarea regulilor unei gramatici într-un program scris în Haskell. Pasul al II-lea: Formarea unei echipe compuse din lingviști și programatori în Haskell cu scopul de a crea produse dedicate pieței de instrumente lingvistice șl în special pieței de manuale elctronice interactive. Pasul al treilea: Crearea unui corpus de reguli gramaticale implementate prin combinatori de parsere, acesta fiind un scop pe termen lung.

Scopul pe termen lung:
Să ajungem în posesia unui corpus de reguli care să reprezinte un întreg limbaj natural, descris cu ajutorul unei colecții de combinatori de parsere. O etapă intermediară este descrierea unei limbi străine la nivelul unui manual inttroductiv al acestei limbi și producerea unui manual electronic bazat pe această tehnologie. O colecție de reguli gramaticale exprimate prin combinatori de parsere provenite de la mai multe exerciții din acest prim manual se transformă într-o grammatică de mari dimensiuni deoarece combinatorii de persere sunt modulari și adaptabili. Practic ei se conformează ideii de proiect software adaptabil așa cum este el prezentat în [Arm 01].

Situația, în prezent

Lingviștii nu sunt familiarizați cu declarațiile arborilor în Haskell. Linviștii nu sunt familiarizați cu folosirea combinatorilor de parsere. Totuși ne putem baza cel puțin pe existența următoarelor competențe: cunoașterea arborilor sintactici și cunoașterea gramaticilor.

Dezvoltarea pînă în prezent

Limbajul Haskell este el însuși un limbaj oferit în domeniul public cu o licență de tip BSD și este șl free-software. În plus, este disponibil pe toate sistemele de operare majoritare de pe piața: Windows ™, Linux ™ , Solaris ™, pentru Mac OS produs de Apple și de asemenea pentru o serie - în creștere - de dispozitive mobile.
Combinatorii de parsere scriși în Haskell au fost folosiți de o serie de autori începând cu Graham Hutton și Erik Meijer [Hut 96], [Hut 98]. În ultimul deceniu au apărut o serie de alte biblioteci, incluzând aici biblioteca de combinatori de parsere cu funcții de semnalizare a

erorilor, Parsec, realizată de Daan Leijen [Lei-01]. Odată cu lansarea Haskell Platform SDK anul 2009, putem găsi în bibliotecile acesteia toate componentele necesare proiectului nostru, inclusiv o versiune a bibliotecii Parsec.
La rândul lor arborii sintactici sunt imediat implementabili în Haskell, deoarece limbajul oferă un instrument specific pentru declararea lur: declarația *data.*
Iar parserele modulare sunt componente software simple și bine documentate de autorii bibliotecilor respective.
Structura algebrică pe care se bazaează combinarea perserelor este monada parserelor – concept din teoria categoriilor – dar lingviștii nu au nevoie să cunoască neaparat aceste detalii: *do notația* din Haskell face ca folosirea monadei parserelor să fie transparentă pentru utilizator iar modulul Monad poate fi importat (inclus) în programe, fiind livrat ca o bibliotecă de sine stătătoare.
Remarcăm și existența unor idei preconcepute dintre care amintim: Ideea că există o unică gramatică a unui limbaj, încă întâlnită printre lingviști este o idee falsă, lucru cunoscut din teoria limbajelor formale.Ideea că limbajul natural evoluează prin adăugarea unor noi situații de comunicare. Se dovedește corectă iar construirea modulară a analizorului sintactic din combinatori de parsere se conformează acestei idei

Arborele sintactic (așa cum a fost realizat pentru un exercițiu din manual)

Pentru o primă implementare am ales un exercițiu dintr-un cunoscut (în România) manual de limbă japoneză [Hon-91]. Pentru a face implementarea se declară întâi tipul arborilor sintactici. Acest lucru este imediat deoarece neterminalii gramaticii se convertesc immediat în *constructorii de date* ai declarației de arbore *data* din Haskell.

```
data Arb = Propozitie Arb Arb
      | Adjdem String
      | Subst String
      | Adj String
      | Subiect Arb Arb
      | PredicatN Arb Arb
      | VerbCopulativ String
      deriving Show
```

Fiecare exercițiu din manual va avea propriul său tip de date corespunzător arborelui său sintactic. Se poate obține însă și un tip comun tuturor exercițiilor declarând împreună toate variantele din instrucțiunile *data*.
Avantaje: Comparat cu codul scris în C cu ajutorul pointerilor șl structurilor, declarația de arbore în Haskell este după cum se vede, mult mai simplă, mai scurtă.

Aspecte privind ordinea cuvintelor în limba Japoneză

Există o deosebire între llimbi europene și limbi asiatice. Limbile europene printre care și limba română folosesc o ordine: subiect, verb, complement. Spre deosebire de acestea, o serie de limbi asiatice – inclusiv limba Japoneză - folosesc frecvent ordinea SOV, subiect, complement, verb. In aceste limbi subiectul este urmat de complemente iar verbul este situat la sfarsitul propozitiei.

Combinatorii de parsere pot implementa rapid orice ordine

Datorita posibilitaăilor lor de a se combina după necesități, ca niște piese de lego, combinatorii de persere pot implementa orice ordine a partilor de propoziție, fără probleme. De exemplu dacă în limba vizată propozițiile respectă ordinea SOV, regula gramaticală va fi scrisă ca mai jos:

```
-- <sentence> -> <subject> <object> <verb>
```

… iar analizorul sintactic monadic modular se scrie imediat în do-notație sub forma:

```
sentence =
  do { x <- subject ;
       y <- object ;
       z <- verb ;
       return (Sentence x y z )
     }
```

În acest exemplu am implementat SOV, ordinea din limba Japoneză. Celelalte cazuri se implementează prin simpla permutare a neterminalilor, respectiv a rândurilor corespunzătoare din program.
Observație: așa cum este scris codul sursă de mai sus, parserul modular *sentence* este scris cu intenția de a se folosi un tip de arbore sintactic conținând un constructor de date numit *Sentence* în locul unuia numit *Propoziție* care apare într-unul din paragrafele anterioare.

Primele exerciții utilizează o sintaxă simplă

Am constatat că primele exerciții din manual, mai simple, utilizează la rândul lor o sintaxă simplă și pot avea ocazional nevoie de reguli mai simple decât cea de mai sus.

```
-- <sentence> -> <subject>  <verb>
```

Corespunzător regulii, analizorul sintactic modular este mai simplu:

```
sentence =
  do { x <- subject ;
       z <- verb ;
       return (Sentence x  z )
     }
```

Exemplul de mai sus corespunde ordinii SOV dar din care complementele (eng. Objects) lipsesc.

O gramatică utilizată de unul dintre exerciții

Unul din modelele de exerciții comune manualelor de limbi străine este numit în limba engleză “fiilling the blancks” sau “create sentences according to the model” (tradus: completați spațiile libere sau formulați propoziții după model.)

Având în vedere gramatica limbajului natural, un lingvist sau un cunoscător de limbaje

formale poate indica ușor subsetul regulilor gramaticale cerute de un anumit exercițiu. Un astfel de set de reguli, creat pentru un exercițiu din lecția 2 dintr-un cunoscut manual [Hon 91] , (ex. B, punctul a, pg 8) este dat în continuare:

```
--  <subst>    -> "ie" | "kiku" | "hon" | "koe"
--  <adj>       -> "ookii" | "chiisai" | "omoshiroi" |  "tsumaranai"
--  <adjdem>   -> "kono" | "sono" | "ano"
-- <subiect>   -> <adjdem> <subst> "wa"
-- <predicat>   -> <adj> <vbcopulativ>
-- <vbcopulativ>   -> "desu" | "dewa arimasen"
-- <propozitie>    -> <subiect> <predicat>
```

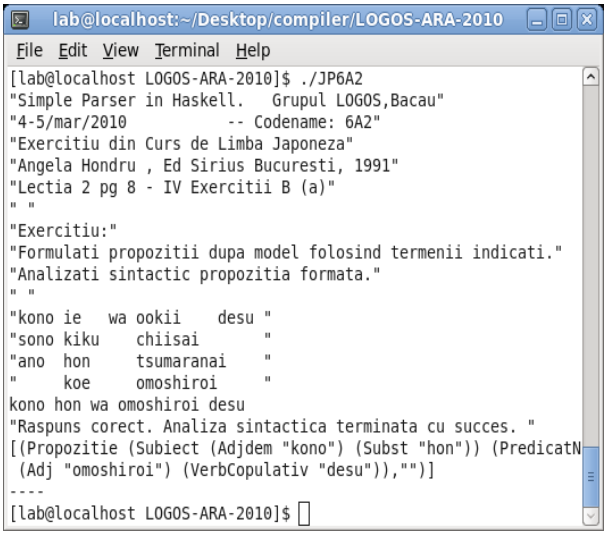
Implementarea exemplului

Regulile gramaticale de mai sus sunt în cele ce urmează transformate în program Haskell, folosind combinatori de parsere (printre care *symbol* – transformă un string în parserul care așteaptă și acceptă acel string), combinatori proveniți din biblioteca ParseLib. Alte biblioteci pot fi de asemenea folosite (ex: Parsec prezentată în [Lei 01]) .

```
--  <subst>  -> "ie" | "kiku" | "hon" | "koe"
subst    = do { symbol "ie" ;
                return (Subst "ie") ;
              }
           +++
           do { symbol "kiku" ;
                return (Subst "kiku") ;
              }
           +++
           do { symbol "hon" ;
                return (Subst "hon") ;
              }
           +++
           do { symbol "koe" ;
                return (Subst "koe") ;
              }

--  <adj>    -> "ookii" | "chiisai" | "omoshiroi" | "tsumaranai"
adj    = do { symbol "ookii" ;
              return (Adj "ookii") ;
            }
         +++
         do { symbol "chiisai" ;
              return (Adj "chiisai") ;
            }
         +++
         do { symbol "omoshiroi" ;
              return (Adj "omoshiroi") ;
            }
         +++
         do { symbol "tsumaranai" ;
              return (Adj "tsumaranai") ;
            }

--  <adjdem> -> "kono" | "sono" | "ano"
adjdem = do { symbol "kono" ;
```

```
                return (Adjdem "kono") ;
              }
            +++
            do { symbol "sono" ;
                 return (Adjdem "sono") ;
               }
            +++
            do { symbol "ano" ;
                 return (Adjdem "ano") ;
               }

-- <subiect> -> <adjdem> <subst> "wa"
subiect = do { a <- adjdem ;
               s <- subst ;
               symbol "wa" ;
               return (Subiect a s) }

-- <predicat> -> <adj> <vbcopulativ>
predicat = do { a <- adj;
                vbc <- vbcopulativ;
                return (PredicatN  a vbc); }

-- <vbcopulativ> -> "desu" | "dewa arimasen"
vbcopulativ = do { symbol "desu" ;
                   return (VerbCopulativ "desu"); }
                 +++
                 do { symbol "dewa" ;
                      symbol "arimasen";
                      return (VerbCopulativ "desu"); }

-- <propozitie>  -> <subiect> <predicat>
propozitie =
  do { x <- subiect ;
       y <- predicat ;
       return (Propozitie x y )    }
```

După rescrierea regulilor gramaticale în Haskell utilizând do-notația și combinatorii de parsere și compilare, codul obiect obținut este link-editat împreună cu restul bibliotecii de combinatori de parsere, rezultând în final un program executabil binar. Am folosit compilatorul GHC, așa cum se vede în imaginea următoare.



Programul principal, are următorul cod sursă:

The main Haskell program was:

```
module Main where
import Prelude hiding (read)
import Monad
import Data.Char

-- Portions  from "Practica interpretarii monadice"
-- Dan Popa, MatrixRom, 2009
-- Parser combinators :  Graham Hutton and Erik Meijer

-------------
main ::IO()
main =
  do { print "Simple Parser in Haskell.Grupul LOGOS,  Bacau"   ;
       print "4-5/mar/2010          -- Codename: 6A2" ;
       print "Exercitiu din Curs de Limba Japoneza";
       print "Angela Hondru, Ed Sirius Bucuresti, 1991" ;
       print "Lectia 2 pg 8 - IV Exercitii B (a)" ;
       print " ";
       print "Exercitiu:" ;
       print "Formulati propozitii dupa model folosind termenii indicati." ;
       print "Analizati sintactic propozitia formata.";
       print " ";
       print "kono ie   wa ookii    desu " ;
       print "sono kiku    chiisai       " ;
       print "ano  hon     tsumaranai    " ;
       print "       koe     omoshiroi     "  ;
       raspuns <- getLine ;
       let rezultat =  parse propozitie raspuns  in
       if (null rezultat)
        then do { print "Raspuns incorect in raport cu cerintele." ;   return ();    }
       else do { print "Raspuns corect. Analiza sintactica terminata cu succes. "  ;
                 print rezultat ;
                 putStrLn "----";
                 return ()
               } ;
       return ()
       }
```

Iar la executarea sa, ca urmare a fazei de analiză sintactică, programul oferă și reprezentarea arborelui sintactic al textului (propoziției) dat(e) ca răspuns, (în cazul în care acest text trece integral de analiza sintactică), sau lista vidă (cu semnificația de răspuns incorect) în caz contrar.

Continuări posibile

Crearea de echipe mixte din lingviști și programatori în Haskell. Transformarea unui set mai vast de exerciții și producerea analizoarelor sintactice modulare. Producerea de cărți electronice interactive și de compact discuri dedicate învățării limbilor străine. Investigarea problemelor suplimentare care pot să apară. Implementarea în produse a bibliotecilor de parsere mai noi, printre care Parsec, recent inclusă (din 2009) în “The Haskell Platform” SDK.

Vocabularul folosit în exemplul anterior

Pentru a face textele de mai sus ceva mai accesibile cititorilor care nu cunosc limba japoneză includem un vocabular minimal, care cuprinde termenii folosiți în exemplul anterior, inclusiv traducerile lor în engleză:

-- Jp/En vocabulary:

-- <subst>
-- ie = home
-- kiku = chrysanthenum
-- hon = book
-- koe = voice

-- <adj>
-- ookiii = big
-- chiisai = small
-- omoshiroi = interesting
-- tsumaranai = not interesting

-- <adjdem>
-- kono = this (close to the speaker)
-- sono = this (close to the interlocutor)
-- ano = that (from the distance)

-- <vbcopulativ>
-- desu = to be
-- dewa arimasen = not to be

-- Jp/Ro vocabulary:

-- <subst>
-- ie = casa
-- kiku = crizantema
-- hon = carte
-- koe = voce

-- <adj>
-- ookiii = mare
-- chiisai = mic
-- omoshiroi = interesant
-- tsumaranai = neinteresant

-- <adjdem>
-- kono = acest/aceasta (de linga mine)
-- sono = acest/aceasta (de lina tine)
-- ano = acel/acea (de la distanta)

-- <vbcopulativ>
-- desu = a fi
-- dewa arimasen = a nu fi

Concluzii

Un set de combinatori de parsere implementați în limbajul funcțional Haskell constituie un bun instrument pentru a programa exerciții lingvistice folosind un computer dotat cu o implementare Haskell, deoarece oferă posibilitatea de a produce rapid analizoare sintactice și interpretoare modulare, adaptabile prin inserarea de module. Metoda de lucru poate fi înțeleasă și de lingviști, aspectele categoriale ale implementării, monada parserelor, operatorii bind și return fiind ascunși de macrodefinițiile *do-notației*.

Bibliografie

[Aab-96] Aaby, A. Anthony; *Haskell Tutorial* http://www.cs.wwc.edu/~cs_dept/KU/PR/Haskell.html

[Arm01] Armour Philip: *The business and software: Zeppelins and jet planes: a methaphor for modern software projects*. Comm. Of ACM, 44(10):13-15 Oct.2001

[Aho07] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, *Compilers Principles, Techniques, & Tools*, sec.ed.2007, Pearson Education

[Hon 91] Hondru, Angela – CURS DE LIMBA JAPONEZĂ fără profesor – Sirius Publishing House, Buc. 1991

[Hut 96] Hutton, Graham; Meijer Errik; Monadic Parser Combinators - "Technical report NOTTCS-TR-96-4" Dept.Comp.Sci. Univ. Nottingham - 1996

[Hut 98] Hutton, Graham; Meijer,Erik ; Monadic Parsing in Haskell – Journal of Functional Programming 8 (4): 437-444, july 1988

[Lei 01] Leijen, Daan – Parsec a fast combinator parser library, Univ. of Utrecht, Dept. of Comp.Sci, Utrecht, The Nederlands

[Pey-02] Peyton Jones, Simon (editor); *Haskell 98 Languages and Libraries – The revised Report*, Cambridge Univ., Sept.2002

[Pop 07] Popa, Dan – Introducere in Haskell 98 prin exemple – EduSoft Publishing House, Bacău, 2007

[Pop 08] Popa, Dan – Practica interpretarii monadice – Matrix Rom Publishing House, Bucharest, 2008

http://www.haskell.org/haskellwiki/Combinatori_de_parsere