Parsing and interpretation

Computational Semantics with Functional Programming

Jan van Eijck (CWI, Amsterdam) & Christina Unger (UiL-OTS, Utrecht)

LOT Summer School
Leiden, June 2009

Outline

Recognizing and parsing simple context-free languages

What is recognition and parsing?

Given a grammar, there are two ways to answer the question, whether a certain string is in the language generated by that grammar:

- yes or no (recognition)
- yes or no, plus how it was derived (parsing)

Recognizing a very simple language

A very simple context-free language is the following screaming language:

$$\mathbf{E} \longrightarrow \textit{argh}! \mid a\,\mathbf{E}$$

Examples: *argh*!, *aargh*!, *aaaaaaaaaargh*!

Exercise: Write a recognizer for it.

```
recognize :: String -> Bool
recognize []        = False
recognize s@(x:xs) = (s == "argh!")
                  || (x == 'a' && recognize xs)
```

From recognition to parsing

A parser should also take a string as input. However, it should not give a Boolean as output but a list of parse trees.

Parse trees represent the structure of the expression, i.e. tell us how it was built.

- If the result is an empty list, the parse failed.
- If it is a singleton list, there is a unique parse.
- Otherwise, the input has more than one parse (i.e. is ambiguous).

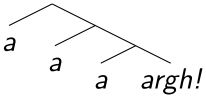
Talking about trees

Trees are either leaves with lexical information, or nodes dominating a list of trees.

```
data Tree a = Leaf a | Branch [Tree a]
```

Example: *aaaargh!*

Tree:




```
aaaargh :: Tree String
aaaargh = Branch [Leaf "a",
                  Branch [Leaf "a",
                          Branch [Leaf "a",
                                  Leaf "argh!"]]]
```

Exercise

Exercise: Write a function

```
string2length :: Tree String -> Tree Int
```

that takes a tree with string leaves as input and replaces all those strings by their length.

Example:

```
(Branch [Leaf "Haskell",Branch [Leaf "is",Leaf "lazy"]])
⇝ Branch [Leaf 7,Branch [Leaf 2,Leaf 4]]
```

Exercise: Write a parser for our screaming language.

```
parse :: String -> [Tree String]
parse []        = []
parse s@(x:xs) = [Leaf   "argh!"        | s == "argh!"]
              ++ [Branch [Leaf "a",t] | t <- parse xs,
                                          x == 'a']
```

Parsers and parse trees

Parsers (more generally)

A parser scans a list of tokens (of type `a`) and tries to construct parse objects (of type `b`) from a prefix of the input list, leaving the remainder for further processing.

```
type Parser a b = [a] -> [(b,[a])]
```

- Again, if the result list is empty, the parse failed.
- If it contains one or more pairs `(parse-object,[])`, the input is in the language generated by the grammar and it has the structure encoded by `parse-object`.
- A pair `(partial-parse-object,[token])` is the result of a partial parse.

Parse trees

We want the parse objects to be trees. We label the nodes of the trees with syntactic information.

```
data ParseTree a b = Leaf a | Branch b [ParseTree a b]
```

Our terminals and labels will be strings, so we will consider parsers of type `Parser String String`, i.e.:

```
[String] -> [(ParseTree String String,[String])]
```

Parsing a natural language fragment

A natural language fragment

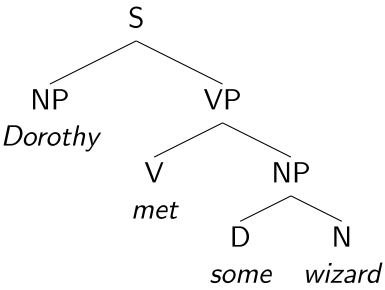
Here is a simple context-free grammar for a very small fragment of English:

$$
\begin{array}{rcl}
\textbf{S} & \longrightarrow & \textbf{NP} \ \ \textbf{VP} \\
\textbf{NP} & \longrightarrow & \textit{Alice} \mid \textit{Dorothy} \mid \textbf{D} \ \ \textbf{N} \\
\textbf{VP} & \longrightarrow & \textit{smiled} \mid \textit{laughed} \mid \textbf{V} \ \textbf{NP} \\
\textbf{D} & \longrightarrow & \textit{every} \mid \textit{some} \mid \textit{no} \\
\textbf{N} & \longrightarrow & \textit{dwarf} \mid \textit{wizard} \\
\textbf{V} & \longrightarrow & \textit{met} \mid \textit{liked}
\end{array}
$$

Some of the sentences we can build:

- *Some dwarf laughed.*
- *Dorothy met Alice.*
- *No wizard liked every dwarf.*

Example parse tree




```
data ParseTree a b = Leaf a | Branch b [ParseTree a b]
```

```
Branch "S" [Leaf "Dorothy",
            Branch "VP" [Leaf "met",
                         Branch "NP" [Leaf "some",
                                      Leaf "wizard"]]]
```

Building parsers

Input to our parsers will be a list of strings.

```
Parsing> words "Dorothy met some wizard"
["Dorothy","met","some","wizard"]
```

We will build parsers that directly correspond to the grammar rules. Our building blocks will be:

- elementary parsers (for parsing terminal strings)
- parser combinators (for $\longrightarrow$ and $|$)

Elementary parsers

Two elementary parsers are parsers that always succeed or always fail.

```
succeed :: b -> Parser a b
succeed r xs = [(r,[])]

failp :: Parser a b
failp xs = []
```

Elementary parsers

For parsing single tokens in our input list, we define the following parser.

```
symbol :: Eq a => a -> Parser a a
symbol s []                    = []
symbol s (x:xs) | s == x       = [(x,xs)]
                | otherwise = []
```

Parsing words

Now we can parse the terminal string *Alice* with the parser

`symbol "Alice"`,

the string *laughed* with the parser

`symbol "laughed"`,

and so on. But we cannot yet handle rules like $\mathbf{N} \longrightarrow \mathit{dwarf} \mid \mathit{wizard}$.
For that we need means for combining parsers into more complex parsers, namely a *dwarf*-parser and a *wizard*-parser into an N-parser.

Parser combinators

Parser combinators are functions that combine parsers into a new parser, or transform a parser into a different parser.

For choice (|), we define a parser combinator <|>, that takes two parsers as arguments and returns a new parser that recognizes everything that either one of the parsers recognizes.

```
(<|>) :: Parser a b -> Parser a b -> Parser a b
(p1 <|> p2) xs = p1 xs ++ p2 xs
```

Now we can define a parser for the rule **N** $\longrightarrow$ *dwarf* | *wizard*.

```
nParser = symbol "dwarf" <|> symbol "wizard"
```

Let us look at cases of nonterminals on the right-hand side of rules, e.g. **NP**⟶**D N**. To build a parser for this, we need sequential composition of parsers. Its general form is:

```
(<*>) :: Parser a b -> Parser a b -> Parser a b
(p <*> q) xs = [ (combine r1 r2,zs) | (r1,ys) <- p xs,
                                       (r2,zs) <- q ys ]
```

Let us assume we simply return strings as parse objects. Then we can define <∗> as follows:

```
(<*>) :: Parser a String -> Parser a String -> Parser a String
(p <*> q) xs = [ (r1 ++ r2,zs) | (r1,ys) <- p xs,
                                 (r2,zs) <- q ys ]
```

Parser combinators

Now, an NP-parser for the rule **NP** $\longrightarrow$ *Alice* | *Dorothy* | **D N** can be implemented like this:

```
npParser =  symbol "Alice"
        <|> symbol "Dorothy"
        <|> (dParser <*> nParser)
```

Parsing the grammar 1

```
sParser :: Parser String String

sParser  =  npParser <*> vpParser

nParser  =  symbol "dwarf"  <|> symbol "wizard"

dParser  =  symbol "every"  <|> symbol "some"
        <|> symbol "no"

npParser =  symbol "Alice"  <|> symbol "Dorothy"
        <|> (dParser <*> nParser)

vpParser =  symbol "smiled" <|> symbol "laughed"
        <|> (vParser <*> npParser)

vParser  =  symbol "liked"  <|> symbol "met"
```

Building parse trees

Up to now, we have built parsers of type `Parser String String`, i.e. they do not yet give parse trees as a result.

In order to get parse trees, we will do postprocessing on the results of a parse. For that we introduce the following parser combinator:

```
(<$>) :: (a -> b) -> Parser c a -> Parser c b
(f <$> p) xs = [ (f r,ys) | (r,ys) <- p xs ]
```

Example:

```
alice :: Parser String Int
alice = length <$> symbol "Alice"
```

Building parse trees

If the symbol we parse is a terminal, the parser should produce a leaf tree.

```
symbolT :: Eq a => a -> Parser a (ParseTree a b)
symbolT s = (\ x -> Leaf x) <$> symbol s
```

For convenience, we define the following type synonym:

```
type PARSER a b = Parser a (ParseTree a b)
```

Building parse trees

For more than one symbol on the right-hand side of a rule, we want the parser to produce a branching tree.

```
parseAs :: b -> [PARSER a b] -> PARSER a b
parseAs label ps = (\ xs -> Branch label xs) <$> collect ps
```

The combinator `collect` collects the result of a list of parses operating one after another.

```
collect :: [Parser a b] -> Parser a [b]
collect []     = succeed []
collect (p:ps) = p <:> collect ps
```

```
(<:>) :: Parser a b -> Parser a [b] -> Parser a [b]
(p <:> q) xs = [ (r:rs,zs) | (r, ys) <- p xs,
                             (rs,zs) <- q ys ]
```

Parsing the grammar 2

```
s,np,vp,d,n,v :: PARSER String Char

s  =  parseAs 'S' [np,vp]

np =  symbolT "Alice"  <|> symbolT "Dorothy"
  <|> parseAs 'N' [d,n]

d  =  symbolT "every"  <|> symbolT "some" <|> symbolT "no"

n  =  symbolT "dwarf"  <|> symbolT "wizard"

vp =  symbolT "smiled" <|> symbolT "laughed"
  <|> parseAs 'V' [v,np]

v  =  symbolT "liked"  <|> symbolT "met"
```

Example

```
Parsing> s (words "Dorothy met some wizard")
 [([.'S' ["Dorothy",[.'V' ["met",[.'N' ["some","wizard"]]]]]],[])]
```

Adding features

Adding features

Adding a feature mechanism to a context-free grammar boils down to replacing rules of the form $A \longrightarrow B\ C$ by rules of the form $A_f \longrightarrow B_g\ C_h$, with f, g, h feature sets whose shape is determined by some feature handling mechanism.

E.g., the rule $\mathbf{S} \longrightarrow \mathbf{NP}\ \mathbf{VP}$ is replaced by:

$$\begin{array}{lll} \mathbf{S}_{\emptyset} & \longrightarrow & \mathbf{NP}_{\{\mathrm{Sg}\}}\ \mathbf{VP}_{\{\mathrm{Sg}\}} \\ \mathbf{S}_{\emptyset} & \longrightarrow & \mathbf{NP}_{\{\mathrm{Pl}\}}\ \mathbf{VP}_{\{\mathrm{Pl}\}} \end{array}$$

So we get:

- *Dorothy laughs.*
- * *Dorothy laugh.*
- *All wizards laugh.*
- * *All wizards laughs.*

Adding features

We implement a mix of feature assignment and feature compatibility check.

For convenience, we put all features in one datatype.

```
data Feat = | Masc | Fem   | Neutr | Sg | Pl
            | Fst  | Snd   | Thrd
            | Nom  | Acc   | Infl  | Wh
```

From strings to categories

Instead of considering words as strings, we implement them as categories, including information about their features and subcategorization properties.

```
data Cat = Cat Phon CatLabel [Feat] [Cat]

type Phon     = String
type CatLabel = String
```

Examples: `lexicon ::  String -> [Cat]`

- `lexicon "alice"  = [Cat "alice"  "NP" [Thrd,Fem,Sg] [] ]`
- ```
  lexicon "helped" = [Cat "helped" "VP" [Tense]
                        [Cat "_" "NP" [Acc]] ]
  ```
  ```

Feature compatibility

The function combine combines the feature lists of two categories. Failure is indicated by [].

```
combine :: Cat -> Cat -> [[Feat]]
combine cat1 cat2 = [ feats | length (gender feats) <= 1,
                              length (number feats) <= 1,
                              length (person feats) <= 1,
                              length (gcase  feats) <= 1,
                              length (tense  feats) <= 1  ]
  where feats = (nub . sort) (fs cat1 ++ fs cat2)
```

Two categories agree if the attempt to combine them does not yield [].

```
agree :: Cat -> Cat -> Bool
agree cat1 cat2 = not (null (combine cat1 cat2))
```

Feature assignment

Some syntactic rules will assign a new feature to a category. E.g. the rule that combines a subject with a predicate will assign the feature Nom to the subject.

```
assign :: Feat -> Cat -> [Cat]
assign f c@(Cat s l fs cs) =
           [Cat s l fs' cs | fs' <- combine c (Cat "" "" [f] [])]
```

Assignment will fail if the category already has an incompatible feature.

Parsing with categories

```
parseNP :: PARSER Cat Cat
parseNP = leafP "NP" <|> npRule
```

```
leafP :: CatLabel -> PARSER Cat Cat
leafP label []     = []
leafP label (c:cs) = [ (Leaf c,cs) | catLabel c == label ]
```

```
npRule :: PARSER Cat Cat
npRule xs = [ (Branch (Cat "_" "NP" fs []) [det,cn],zs) |
              (det,ys) <- parseDET xs,
              (cn, zs) <- parseCN  ys,
               fs      <- combine (t2c det) (t2c cn),
              agreeC det cn ]
```

Parsing with categories

```
parseSent :: PARSER Cat Cat
parseSent = sRule

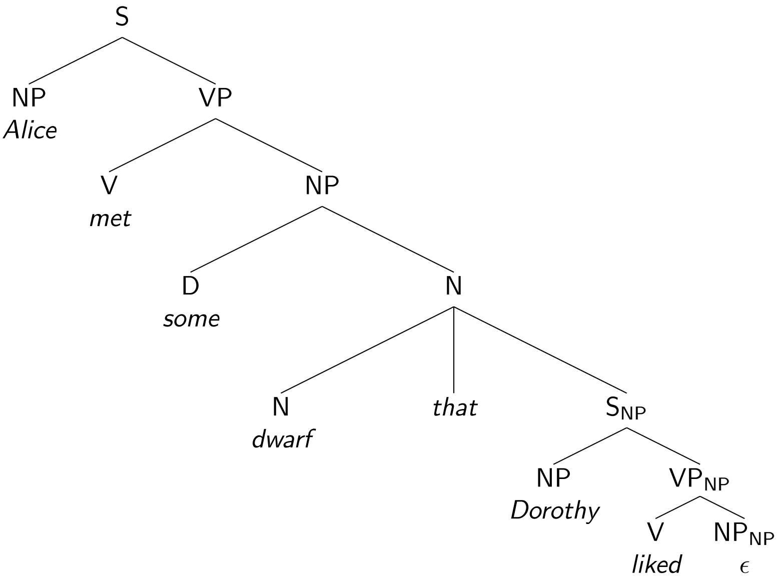
sRule :: PARSER Cat Cat
sRule xs = [ (Branch (Cat "_" "S" [] []) [np',vp],zs) |
              (np,ys) <- parseNP xs,
              (vp,zs) <- parseVP ys,
               np'    <- assignT Nom np,
              agreeC np vp,
              subcatList (t2c vp) == [] ]
```

Adding extraction

Grammars with extraction

$$
\begin{array}{rcl}
\mathbf{S} & \longrightarrow & \mathbf{NP}\ \ \mathbf{VP} \\
\mathbf{S_{NP}} & \longrightarrow & \mathbf{NP_{NP}}\ \mathbf{VP} \mid \mathbf{NP}\ \mathbf{VP_{NP}} \\
\mathbf{NP} & \longrightarrow & \textit{Alice} \mid \textit{Dorothy} \mid \mathbf{D}\ \ \mathbf{N} \\
\mathbf{NP_{NP}} & \longrightarrow & \epsilon \\
\mathbf{VP} & \longrightarrow & \textit{smiled} \mid \textit{laughed} \mid \mathbf{V}\ \mathbf{NP} \\
\mathbf{VP_{NP}} & \longrightarrow & \mathbf{V}\ \mathbf{NP_{NP}} \\
\mathbf{D} & \longrightarrow & \textit{every} \mid \textit{some} \mid \textit{no} \\
\mathbf{N} & \longrightarrow & \textit{dwarf} \mid \textit{wizard} \mid \mathbf{N}\ \textit{that}\ \mathbf{S_{NP}} \\
\mathbf{V} & \longrightarrow & \textit{met} \mid \textit{liked}
\end{array}
$$

Example


S
NP
Alice
VP
V
met
NP
D
some
N
N
dwarf
that
S_NP
NP
Dorothy
VP_NP
V
liked
NP_NP
ϵ

Stack parsers for extraction

For handling extractions, we enrich our parsers with a stack of 'extracted material'.

```
type StackParser a b = [a] -> [a] -> [(b,[a],[a])]
type SPARSER     a b = StackParser a (ParseTree a b)
```

We define adapted versions of the parser combinators, that pass around the stack, as well as stack operations for pushing new items on the stack and popping the top item from a stack.

```
push :: Cat -> SPARSER Cat Cat -> SPARSER Cat Cat
push c p stack = p (c:stack)
```

```
pop :: CatLabel -> SPARSER Cat Cat
pop c []     xs                    = []
pop c (u:us) xs | catLabel u == c = [(Leaf u, us, xs)]
                | otherwise        = []
```

Example

Relative clauses consist of a relative plus a sentence with an NP-gap. This is created by pushing a gap of category NP on a parser for sentences.

```
relR  :: SPARSER Cat Cat
relR us xs = [(Branch (Cat "_" "COMP" fs []) [rel,s],ws,zs) |
               (rel,vs,ys) <- leafPS "REL" us xs,
                fs          <- [fs (t2c rel)],
                gap         <- [Cat "#" "NP" fs []],
               (s,ws,zs)    <- push gap prsS vs ys ]
```

An NP can be parsed by popping an NP from the extraction stack.

```
prsNP :: SPARSER Cat Cat
prsNP = leafPS "NP" <||> npR <||> pop "NP"
```

The final parsing function

```
parses :: String -> [ParseTree Cat Cat]
parses str = let ws = lexer str
             in  [ s | catlist   <- collectCats lexicon ws,
                       (s,[],[]) <- prsTXT [] catlist
                                 ++ prsYN  [] catlist
                                 ++ prsWH  [] catlist ]
```

- `lexer` preprocesses the string (removes punctuation marks, maps capital letters to lower ones, and so on)
- `collectCats` looks up the words in a database (like `lexicon`) of type `String -> [Cat]`

Testing: `testSuite1`, `testSuite2`

Adding semantics

Defining logical forms

First, we define logical forms `LF`.

```
data LF = Rel String [Term]
        | Eq   Term Term
        | Neg  LF
        | Impl LF LF
        | Equi LF LF
        | Conj [LF]
        | Disj [LF]
        | Qt GQ Abstract Abstract

data Term     = Const String | Var Int
data GQ       = Sm | All | Th | Most | Many | Few
data Abstract = MkAbstract Int LF
```

Translating trees into LFs

Then we map parse trees to logical forms by matching every parse rule with a logical form translation rule.

Examples:

```
trS :: ParseTree Cat Cat -> LF
trS (Branch (Cat _ "S" _ _) [np,vp]) = (trNP np) (trVP vp)

trNP :: ParseTree Cat Cat -> (Term -> LF) -> LF
trNP (Leaf   (Cat "#"  "NP" _ _))           =  p -> p (Var 0)
trNP (Leaf   (Cat name "NP" _ _))           =  p -> p (Const name)
trNP (Branch (Cat _    "NP" _ _) [det,cn]) = (trDET det) (trCN cn)
```

Translating trees into LFs

```
process :: String -> [LF]
process string = map transS (parses string)
```

Finally, we need one more step for interpreting these logical forms.