

Abigail Swain
May 7, 2007

Earley Parser

With this project, I wished to accomplish a small piece of the natural language processing problem. Even the first couple chapters of Terry Winograd's *Understanding natural language*, in which he describes his awesome program for understanding natural language, helped to convince me of the enormity of this problem. I undertook to write a program that would allow me to tackle a small part of that undertaking. My program parses simple sentences into hierarchies of phrasal and lexical categories that represent phrase-structure trees. This program has three main parts: a grammar, containing phrase-structure rules, a dictionary, containing lexemes and their parts of speech, and the parser machine itself.

Design: Following the description given by Jurafsky and Martin in *Speech and Language Processing* and with the help of my advisor, Dr Brett Kessler, I wrote a program in Perl which implements the Earley algorithm to parse a sentence. I designed a number of objects, each representing either a part of the parser or a part of the grammar and dictionary which provide necessary inputs to the parser. The executable `parser.pl` file controls the mechanism of the parser, first gathering information from the user such as where dictionary and grammar information is stored and then creating a new Parser object containing this information. The Parser object is called on to parse a sentence provided by the user, which creates both a new Sentence object holding each of its constituent words and a new Chart object to hold the data stored at each stage of parsing.

The chart is an array of one more than the number of words in the sentence ($N + 1$) columns. Each column contains information about the progress of the parse at a certain point in a sentence (before, between or after the words.) This information is stored in the form of State objects, each containing information about a phrasal rule that may be part of a successful parse (a Rule object) and information about the progress of the parsing up until that point. The array of State objects within each column behaves like an agenda: each state is processed in order from top to bottom, starting at the top of the first column and ending at the bottom of the last. A function within the Parser class decides how each State needs to be processed as it comes to the top of the agenda, sending it to different methods in the State class depending on what category in the Rule object has yet to be incorporated into the parse.

These methods, the scanner, the predictor, and the completer, add new states to the chart in order that they be put on the agenda for future consideration. The scanner advances a State's phrase-

structure rule over a lexical category by adding a State associating that category with the appropriate word in the sentence. The predictor looks at the next phrasal category that has yet to be parsed, and adds State objects for each of the rules provided in the grammar which have that category as a root. The completer advances a State's rule over a phrasal category using information about the completeness of that category from previous States in the Chart.

When all the States in each of the $N + 1$ columns have been read and processed appropriately, the Parser returns the completed Chart, which now contains enough information to construct the phrasal structure of the sentence. The executable file is in charge of printing out the entire chart and the nested-parentheses representation of the parse which can be extracted from the Chart. Please see Appendix B for an example of the interaction between the user and the program.

Dictionary and Grammar: The dictionary is represented by a Dictionary object, which contains an array of Entry objects, each holding information about a single word and the list of lexical categories associated with that word. These objects are created from a text file containing entries listed on separate lines. A dictionary entry must be of the format:

```
spelling:lexical_category1,lexical_category2, as in:  
bike:singular_noun,transitive_verb.
```

The grammar is represented by a Grammar object, which contains an array of Rule objects, each holding information about a phrase-structure rule, with a phrasal category as a root and one or more subcategories. Like the dictionary, the information in a Grammar object is retrieved from a text file containing rules listed on separate lines. The program looks for a rule of the format:

```
phrasal_category-->subcategory1,subcategory2, as in:  
S-->NP_singular,VP_singular
```

Limitations: The actual parsing machine of this program is relatively robust. The limitations of the program are mostly due to the limited amount of information it will accept in representing the Grammar and the Dictionary. For one thing, the parser will only work if the labels for the phrasal and lexical categories are designed to be one dimensional; these labels are treated as strings and are not further analyzed for, say, feature values. I began an attempt to modify the program to work with categories that contain feature variables and values, but I reverted to the original due to time constraints. I considered the following design for feature-based rules in the grammar:

```
S->NP[case=nom;number= $\beta$ ],VP[number= $\beta$ ], and for feature-based dictionary entries:  
bike:N[case=nom;number=sing],N[case=acc;number=sing]
```

When the parser reads these lines in, it would have to store the values for case and number in separate instance variables of the Rule and Entry objects, respectively. Modifications to the Parser, State, and Chart classes would also be necessary to have the program work with rules and entries of this kind; for one thing, the scanner, predictor, and completer would have to be reprogrammed to determine the compatibility of categories within a state along each feature dimension.

In the program's current state, given one-dimensional category labels, the number one reason why the parser will fail to parse a perfectly passable sentence is a problem with the content and consistency of the grammar and dictionary. If a phrasal rule integral to the structure of the sentence is not provided in the grammar, no parse will be found. Similarly, if a word in the sentence is not associated in the dictionary with the lexical category it represents in the sentence, no parse will be found. The parser will also fail if a subcategory of a rule in the grammar does not also exist either as a lexical category in the dictionary or as a phrasal category (or root) in the grammar. In this case, the program will not be able to expand this category for further parsing.

If a word in a given sentence is not found in the Dictionary as created from the original text file, the program *does* allow a user to add further entries to the Dictionary object (and to the original text file.) In this situation the program prompts the user to provide a lexical category for that word (only one category is accepted at this time, though a word may have several.) This feature allows for ease in expanding the dictionary according to the user's specific lexical needs, and adds to the flexibility of the parser, since it would otherwise fail to parse a sentence with an unknown word.

The grammar file, `grammar.txt`, that I developed for testing this program, seen in Appendix A, contains a good number of phrase structure rules for basic sentence structures, but is by no means comprehensive of English syntax. My program would be well complemented by a larger and more complete grammar and dictionary.

Testing: The executable file `test_parser.pl` is designed to read in a text file containing sentences and the nested-parentheses parses the user expects to be produced for them. The following lines could be found in such a file:

```
he bikes=S(Npnom3(ProNomSing(he))VP3(Vintr3(bikes)))
he bike=null
```

Note how the user expects a null result in the second line, because the sentence is ungrammatical.

Once this file is read in, `test_parser.pl` calls on the parsing machine to return its parse for each of the sentences and compares the expected and actual parses. The testing program then returns

information about which cases pass and which fail. Below is an example of the results this program would return. The two cases seen above both pass. The third case fails because the expected parse does not match the parses the program returns (I designed this case specifically to fail); the parses returned by the program are printed out so they may be compared to the expected value.

```
PASS: he bikes, s(npnom3(pronom3sing(he))vp3(vintr3(bikes)))
PASS: he bike, null
FAIL: The cyclist raced the winner from the previous year
      Expected: bogus(parse)
      Actual 1: s(npnom3(detsing(the)nsing(cyclist))vp3(vtr3(raced)
      npacc(detsing(the)nsing(winner))pp(p(from)npacc(detsing(the)
      adjp(adj(previous))nsing(year))))
      Actual 2: s(npnom3(detsing(the)nsing(cyclist))vp3(vtr3(raced)
      npacc(detsing(the)nsing(winner))pp(p(from)npacc(detsing(the)
      adjp(adj(previous))nsing(year))))))
```

```
2 passed.
1 failed.
```

The testing program will compare an expected parse with all possible parses provided by the program (note the two `Actual` cases provided in the example above.) So the following two cases, which show two different expected parses for the same sentence, both pass:

```
PASS: he rides bikes with ease,
s(npnom3(pronom3sing(he))vp3(vtr3(rides)npacc(nplural(bikes))pp(p(with)npacc(nmass(ease))))))
PASS: he rides bikes with ease,
s(npnom3(pronom3sing(he))vp3(vtr3(rides)npacc(nplural(bikes))pp(p(with)npacc(nmass(ease))))))
```

(The difference between the two parses is whether the prepositional phrase is a subconstituent of the `Npacc` or of the `s`.)

APPENDIX A (Contents of text files developed to test parser program.)

Most of grammar.txt

S-->NPnom,VP
S-->NPnom3,VP3
S-->PP,NPnom,VP
S-->PP,NPnom3,VP3
S-->VP
S-->VP,PP
VP-->Vintr
VP3-->Vintr3
VP-->Vintr,PP
VP3-->Vintr3,PP
VP-->Vtr,NPacc
VP3-->Vtr3,NPacc
VP-->Vtr,NPacc,PP
VP3-->Vtr3,NPacc,PP
VP-->Vditr,NPacc,NPacc
VP-->Vditr,NPacc,to,NPacc
VP3-->Vditr3,NPacc,NPacc
VP3-->Vditr3,NPacc,to,NPacc
VP-->Vditr,NPacc,NPacc,PP
VP-->Vditr,NPacc,to,NPacc,PP
VP3-->Vditr3,NPacc,NPacc,PP
VP3-->Vditr3,NPacc,to,NPacc,PP
NPnom3-->Nmass
NPnom3-->DetSing,Nsing
NPnom-->DetPlural,Nplural
NPnom-->Nplural
NPnom3-->AdjP,Nmass
NPnom3-->DetSing,AdjP,Nsing
NPnom-->DetPlural,AdjP,Nplural
NPnom-->AdjP,Nplural
NPnom3-->Nmass,PP
NPnom3-->DetSing,Nsing,PP
NPnom-->DetPlural,Nplural,PP
NPnom-->Nplural,PP
NPnom3-->AdjP,Nmass,PP
NPnom3-->DetSing,AdjP,Nsing,PP
NPnom-->DetPlural,AdjP,Nplural,PP
NPnom-->AdjP,Nplural,PP
NPnom3-->ProNomSing
NPnom-->ProNomPlural
NPacc-->Nmass
NPacc-->DetSing,Nsing
NPacc-->DetPlural,Nplural
NPacc-->Nplural
NPacc-->AdjP,Nmass
NPacc-->DetSing,AdjP,Nsing
NPacc-->DetPlural,AdjP,Nplural
NPacc-->AdjP,Nplural
NPacc-->Nmass,PP
NPacc-->DetSing,Nsing,PP
NPacc-->DetPlural,Nplural,PP
NPacc-->Nplural,PP
NPacc-->AdjP,Nmass,PP
NPacc-->AdjP,Nplural,PP
NPacc-->DetSing,AdjP,Nsing,PP
NPacc-->DetPlural,AdjP,Nplural,PP
NPacc-->ProAccSing
NPacc-->ProAccPlural
AdjP-->Adj

Selection from dictionary.txt

bike:Vintr,Nsing
biked:Vintr,Vintr3
bikes:Vintr3,Nplural
ride:Vintr,Nsing,Vtr
rode:Vintr,Vintr3,Vtr,Vtr3
rides:Vintr3,Nplural,Vtr3
give:Vditr
gives:Vditr3
gave:Vditr,Vditr3
he:ProNomSing
him:ProAccSing
they:ProNomPlural
them:ProAccPlural
Abigail:Nmass
speed:Nmass,Nplural,Nsing
hill:Nsing
hills:Nplural
near:P
need:Vtr,Nsing
needed:Vtr,Vtr3
needs:Vtr3,Nplural
steep:ADJ
a:DetSing
those:DetPlural
the:DetSing,DetPlural
on:P
to:P,to
vacation:Nsing,Nmass
vacations:Nplural
with:P
ease:Nmass
adam:nmass
loves:vtr3
biking:nmass
she:pronomsing
likes:vtr3
maggie:nmass
rocks:nplural
frenchmen:Nplural
in:p
lincoln:nmass
tires:nplural
of:p
racer:nsing
an:detsing
advantage:nsing
mountains:nsing
mountains:nplural
racers:nplural
encountered:vtr,vtr3
large:adj
france:nmass
from:p
has:vtr3
dangerous:adj
roads:nplural
have:vtr
sections:nplural
that:detsing

APPENDIX B (Command line interface between parser.pl and user. User input is shown in **bold**.)

```
C:\Perl 2007>perl parser.pl
Please specify grammar text file: grammar.txt
Is 'G-->S' the start rule for the specified grammar? (yes/no) yes
please specify dictionary text file: dictionary.txt
Would you like the chart to be printed to a file? (yes/no) yes
Please specify file to which chart should be printed: chart.txt
Would you like the log information to be printed to a file? (yes/no) yes
Please specify file to which log info should be printed: log.txt
What sentence would you like me to parse? he rides a bike
Parse: s(npnom3(prons(he))vp3(vtr3(rides)npacc(detsing(a)nsing(bike))))
Another sentence (or type 'ESC' for exit)
What sentence would you like me to parse? He rides motorcycles with friends
Entry not found for word {motorcycles}.
Please specify part of speech (or enter 'help'): help
    Possible parts of speech are:
    adj
    detplural
    detsing
    nmass
    nplural
    nsing
    p
    proap
    proas
    pronp
    prons
    to
    vditr
    vditr3
    vintr
    vintr3
    vtr
    vtr3
Please specify part of speech for {motorcycles} from above options: nplural
Entry not found for word {friends}.
Please specify part of speech (or enter 'help'): nplural
Parse: s(npnom3(prons(he))vp3(vtr3(rides)npacc(nplural(motorcycles))pp(p(with)npacc(nplural(friends)))))
Parse: s(npnom3(prons(he))vp3(vtr3(rides)npacc(nplural(motorcycles))pp(p(with)npacc(nplural(friends)))))
Another sentence (or type 'ESC' for exit)
What sentence would you like me to parse? Esc
```

(The manual inputs regarding file names seen at the beginning of this text are the same as those that would be used if the program were run with the `--default` flag. This flag tells the program to use the files `grammar.txt` and `dictionary.txt`, and to print information to `chart.txt` and `log.txt`. The tag allows you to avoid some typing. The `--default` flag is accomplishes more or less the same thing for the `test_parser.pl` program, indicating that `grammar.txt` and `dictionary.txt` should be used.)