Massachusetts Institute of Technology
6.863J/9.611J Natural Language Processing, Spring, 2001
Department of Electrical Engineering and Computer Science
Department of Brain and Cognitive Sciences

Handout 8: The 6.863 Earley Parser

1 Introduction

The 6.863 Earley parser will work with any well-defined context-free grammar (CFG), including recursive ones and those containing empty categories. It can use either no lookahead or a one word lookahead, *i.e.* $k = 0$ or $k = 1$. A feature system has been overlaid on the parser, allowing powerful grammars to be created more easily than with a simple CFG. Although this feature system does not by any means implement any reasonable variation of the GPSG theory, it will be referred to as GPSG for historical reasons.

In addition to the parser, the Earley system has Lisp facilities for creating and editing CFG and GPSG grammars, dictionaries, and sentence sets.

2 Loading the System

All of the parser and utility code is Common Lisp, and were all Common Lisps equal, the parser would be machine independent. The files are set to load into the package `USER`. The file `earley.lisp` contains all the necessary code.

3 A Quick Overview

The Earley system stores various types of *rule sets*, such as CFG grammars, GPSG grammars, or sentence sets. These can be edited and used as input for the parser. The parser requires special grammar tables as input, which are compiled from featureless context-free grammars (GPSGs are automatically translated into featureless CFGs).

To look at the rule sets stored, evaluate `(list-all)`. This prints out the name, type, and number of rules in each rule set. There are 6 different types of rule sets: CFG, GPSG, CFG-TABLE, GPSG-TABLE, SENTENCES, and DICTIONARY. The -TABLE types are used as input for the parser. They cannot be edited, but all other types of rule sets can.

3.1 Parsing Sentences

Before we can parse a sentence we need to supply grammar information. The parser requires a DICTIONARY and either a CFG-TABLE or a GPSG-TABLE before it can parse. Parsing is handled through the `p` function. Let's say we have the CFG-TABLE `CT` stored, and also the dictionary `DICT`. We can parse the sentence "John saw Mary" by evaluating

```
>  (p '(JOHN SAW MARY) :grammar 'CT :dictionary 'DICT)
```

which will return a list of valid tree structures. The grammar and dictionary variables are sticky, so now that we've specified the grammar to use, to parse "John walked into the tree" only requires

```
>  (p '(JOHN WALKED INTO THE TREE))
```

Parses are returned as lists of possible tree structures, where each tree structure has the form *(Category SubTree1 SubTree2 ...)*. Thus for the above example we might get back the parse list

```
((ROOT (S (NP (N JOHN))
          (VP (VP (V WALKED))
              (PP (P INTO) (NP (D THE) (NP (N TREE))))))))
```

The full form of the *p* function is

```
(p sentence &key (template '*) grammar dictionary (use-lookahead t)
                 print-states file sentence-set)
```

where `sentence` can either be a list or a string; `:template` specifies which parses are to be accepted; `:grammar` and `:dictionary` specify the grammar and dictionary to use; `:use-lookahead` specifies whether or not to use lookahead if it is available; `:print-states` specifies whether or not to print out all the Earley states at the end of the parse; `:file`, if given, is the name of a file to send all output to; and `:sentence-set` specifies a set of sentences to parse. After a parse has been completed the function `retrieve-parses` with the format

```
(retrieve-parses &optional (template '*))
```

can be used to retrieve the results again. More parsing functions, including ones to parse sets of sentences at one time, are given in the function listing later.

3.2 Editing Rule Sets

If there is a rule set `SD`, a DICTIONARY, it can be listed by evaluating `(list-rules 'SD)`. This will print out all of the rules (dictionary entries) in the set. A template may be specified, so all verbs in `SD` could be listed with `(list-rules 'SD '(* V))`. Likewise, all rules with verb phrases as heads in the CFG `GRAMMAR1` could be listed with `(list-rules 'GRAMMAR1 '(VP **))`.

List-rules does more than just print out rules: it also sets up rule sets for quick editing. Let's say we evaluated `(list-rules 'SD '(* A))` to list out all the adjectives in the dictionary `SD`. Perhaps the result was

```
>  (list-rules 'SD '(* A))
  1.    HARD A
  2.    QUICK A
  3.    RED A
  4.    SLIMY A
  5.    SLOW A
```

We can now use the three functions `ADD`, `DEL`, and `RPL` to edit the dictionary `SD`. To delete "hard" and "slimy" we evaluate `(del 1 4)`. Now, we decide to add "lazy" in their place, so we do `(add '(LAZY A))`, and finally we decide to replace "quick" with "fast" and "slow" with an adverb, "slowly" - `(rpl 2 '(FAST A) 5 '(SLOWLY ADV))`. These three functions work on the last rule set listed with `list-rules`. Thus they permit quick and easy editing of grammars, dictionaries, and test sentences.

3.3 Creating and Removing Rule Sets

To simply remove or add a rule set, we can use the functions `(remove-rule-set name)` and `(add-rule-set name type)`. Then rules can be added or removed from them either by using the previously mentioned editing functions, or through

```
(add-rule name-of-rule-set rule)
;; (add-rule 'GRAMMAR1 '(S ==> NP AUXP VP))

(del-rule name-of-rule-set rule)

(add-rule-list name-of-rule-set rule-list)
;; (add-rule-list 'DICT '((THE D) (AND CONJ)))

(del-rule-list name-of-rule-set rule-list)
```

These will not help us generate the tables the parsers need for running, but only base rule sets. Specialized functions are provided to compile rule sets into -TABLEs. Translating a CFG `GRAMMAR1` to a CFG-TABLE `CT` usable by the parser takes one step, which requires specifying the root node of the CFG and also whether $k = 0$ or $k = 1$. The lookahead table takes little time to compute, so presumably k will be 1. The translation function is `create-cfg-table`, which takes the form

```
(create-cfg-table table-name cfg-name root k)
```

or in our case, with root `ROOT`

```
(create-cfg-table 'CT 'GRAMMAR1 'ROOT 1)
```

The corresponding function for GPSG grammars is `create-gpsg-table`, which takes the same arguments.

3.4 Templates

There are two places templates are used, in retrieving parses and in listing rule sets. They merely provide a filtering system to either limit the types of parse tree structures accepted or limit the types of rules to be printed. Look back to the respective sections for more background information.

Templates follow four basic rules

1. A symbol matches itself or any list with that symbol as it's first element, or car.
2. A list matches another list if each element matches, or if each element before a double asterisk in the template list matches the other list.
3. An asterisk * matches any symbol or list.
4. A double asterisk ** matches any symbol or list unless it appears in a list, in which case it matches any ending to that list.

thus the template `NP` matches `NP` and `(NP (N JOHN))`, the template `(* V)` matches `(JUMP V)`, and the template `(* ==> NP **)` matches any list with `==>` as it's second element and `NP` as it's third element. For instance, if we had just parsed a sentence which 26 different structures, we might want to examine only those with an adjunct prepositional phrase attached to the verb phrase after the object.

```
>  (retrieve-parses '(ROOT (S NP (VP * PP **) **)))
```

3.5 Loading and Saving Rule Sets

Some rule sets can be saved in text files designed to be read back with the standard Lisp `load` function. Unfortunately, because Lisp has problems reading arrays, rule sets of type CFG-TABLE and GPSG-TABLE can not be saved. So even if a text file is used to store a base grammar, the parser tables must be recompiled before use.

To save rule sets to a readable text file, use the function `save-rules` supplying as arguments the name of the file and the names of all the rule sets we wish to save. If only the file name is supplied, all rule sets possible to save are saved. Below two rule sets are saved, deleted, and reloaded. given below.

```
>  (save-rules "~/rules.lisp" 'RULE-SET-1 'RULE-SET-2)
>  (remove-rule-set 'RULE-SET-1)
>  (remove-rule-set 'RULE-SET-2)
>  (load "~/rules.lisp")
```

3.6 Incidental Warnings

Many functions will return incedental messages as they run. These less important messages can be toggled by evaluating `(verbose)`, or can be specified with `(verbose :set nil)` and `(verbose :set t)`. Detailed information on GPSG parsing is printed when `verbose` is turned on.

4 Rule Formats

Dictionary entries and CFG or GPSG rules have certain required formats. A DICTIONARY entry must by a list with the first element the symbol for the word, and all other elements either symbols or lists representing the categories the word can take. Thus sample entries are `(WALK N (V Action + Tense INF))` and `(I (PRO Pers 1st))`. A dictionary without features will

work with a GPSG set of rules, but each word will be featureless. Likewise, features will be ignored when using a GPSG dictionary with a CFG. If a word is entered multiple times (each time with one or more definitions), the union of the sets of definitions will be used.

CFG rules take the format `(HEAD ==> T1 T2 ... Tn)` where n can even be zero, representing an empty category. Each element in the list must be a symbol and the second element must be `==>`. A GPSG rule has GPSG categories substituted for symbols. A GPSG category is either a symbol (like a CFG category) or a list with the category type as it's first element. After the first element come pairs of items, the first being a feature name and the second being a value, either a symbol or a category. Features to be matched for unification can be specified with a value starting with "?". Some valid CFG and GPSG rules are:

```
;; CFG
(NP ==> D N)
(S ==> NP VP)
(NP ==> ) ;; empty category
;; GPSG
((AUX ADV +) ==> AUXADVP)
((AP WH ?a) ==> (AP WH ?a) (VP FIN ?b / (NP WH ?b)))
((NP PL +) ==> (D) (N PL +))
((S FIN ?val) ==> (NP WH -) (AUX FIN ?val) (VP FIN ?val))
```

The parser does not parse GPSGs directly, but instead parses a featureless set of rules and then adds features and performs unification later. This means that parsing may go very quickly for GPSGs, but parse retrieval may take longer.

Sentence sets treat sentences as rules, and each sentence can either be a string or a list of words (symbols). The sentences can be parsed as a group, making multiple tests a bit easier.

5 Debugging Parses

The parsing function `p` will print out the states generated by the Earley parser while it runs if the `:print-states t` argument is given. But this often doesn't provide enough information for debugging GPSG parses. After a parse has been completed the function `states` can be used to get more detailed information about what occurred.

`(states start end &key matching print-parses print-rules)`

`states` takes two arguments which specify the start point and end point of states to be printed. In other words, it only prints information about completed states that build phrases between the starting and ending points. If `:print-rules` is specified to be true, then along with each state a corresponding GPSG rule is printed. If `:print-parses` is true, then all of the phrases that were created are printed for each state. The `:matching` argument is a filter to apply to these phrases. Each of the states is printed with an asterisk preceding it if no phrases were produced during the GPSG unification process.

```
>  (p '(who saw mary) :print-states t)
_ 0  0   *DO* ==> . ROOT $
_ 0  0   ROOT ==> . S
```

```
...

_ 3  3   S-W/NP ==> . S/NP
_ 3  3   S-W/NP ==> . AUX S-NOAUX/NP
_ 3  3   S-W/NP ==> . AUX S/NP
(((ROOT)
   ((QBAR) ((NP WH + PRO +) ((PRO WH +) WHO))
           ((S-W FIN + / NP)
             ((S FIN + / NP OBJ-MOVT -)
               ((NP / NP))
               ((VP FIN +) ((V2 FIN +) SAW)
                           ((NP WH -) ((NAME) MARY))))))))

>  (states 1 3)
_ 1  1   VP ==> V2 NP .
_ 1  1   S/NP ==> NP/NP VP .
* 1  1   R ==> S/NP .
_ 1  1   S-W/NP ==> S/NP .

>  (states 1 3 :print-rules t)
_ 1  1   VP ==> V2 NP .
  Rule: ((VP FIN ?A) ==> (V2 FIN ?A) (NP))
_ 1  1   S/NP ==> NP/NP VP .
  Rule: ((S FIN + / NP OBJ-MOVT -) ==> (NP / NP) (VP FIN +))
* 1  1   R ==> S/NP .
  Rule: ((R) ==> (S FIN + / NP OBJ-MOVT + REL +))
_ 1  1   S-W/NP ==> S/NP .
  Rule: ((S-W FIN ?A / ?B) ==> (S OBJ-MOVT - FIN ?A / ?B))

>  (states 1 3)
_ 1  1   VP ==> V2 NP .
_ 1  1   S/NP ==> NP/NP VP .
* 1  1   R ==> S/NP .
_ 1  1   S-W/NP ==> S/NP .

>  (states 0 1 :print-parses t)
_ 0  0   NP ==> PRO .
((NP WH + PRO +) ((PRO WH +) WHO))
* 0  0   ROOT ==> NP .
```

`states` prints an asterisk in front of a state if that state failed to produce any phrases during the GPSG feature-passing portion of a GPSG parse. Unfortunately `states` can not provide any information on why a rule might have failed. Using the `:print-parses` option lets you see the phrases that result from successfull rule applications, but not failed ones. This

information is available during the parse by parsing with `verbose` on. For instance, we can find out why the starred state `* 1 1 R ==> S/NP .` failed

```
>  (verbose :set t)
T
>  (p '(who saw mary))
Done CFG Parse.
Rule Failure:  (R) ==> (S FIN + / NP OBJ-MOVT + REL +)
Sub Trees:  (S FIN + OBJ-MOVT - / NP)
Rule Failure:  (S FIN + OBJ-MOVT +) ==> (NP WH - / NIL) (VP FIN +)
Sub Trees:  (NP WH + SPECIFIC +) (VP FIN +)
Rule Failure:  (S FIN + OBJ-MOVT + / ?SLASH) ==> (NP WH - / NIL)
                                                 (VP FIN + / ?SLASH)
Sub Trees:  (NP WH + SPECIFIC +) (VP FIN + / NP)
Done Parse Retrieval
```

While this provides the information we need, in a large parse it can be difficult to sift through the entire printout to find the desired failure points.

6 GPSG Rules and Feature Passing

GPSG rules are preprocessed before parsing. Their features are stripped to created a featureless CFG that is used by the parser in the first stage of parsing. Parse trees that are recovered by the initial parse are augmented with features in a second pass that utilizes the original featured rules.

Features are used to refine rules without multiplying out the number of categories. If a rule has in its right hand side a category with a certain feature specification, then whatever subphrase matches that category must have the same feature specification. This can be used to ensure, for instance, that a rule matches only plural NPs and not just NPs in general without having to create a seperate category for plural NPs. Feature values can be passed upwards, so that a phrase inherits feature values from its daughters.

With the feature system it is possible to create ridiculously baroque devices. As much care as possible should be taken to avoid unchecked feature generation; unfortunately the simplicity of the context-free-grammar formalism sometimes necessitates this.

In a GPSG rule, each category can either be a simple category symbol or a list of a category symbol and feature-value pairs. A feature-value pair is a combination of a symbol that names a feature and a value for that feature. Possible GPSG categories are `NP`, `(NP)`, `(NP PLURAL +)` and `(NP PLURAL ?X / VP)`. In these categories the symbols `plural` and `/` are feature names and `+`, `?X` and `VP` are feature values. Features found in the left hand side of a GPSG rule are given to the phrase produced by the rule. Features on the right hand side refine which daughters will match the categories- daughter phrases must contain the features specified. So the rule

`((NP PLURAL +) ==> (DET SING -) ADJ (N PLURAL +))`

will fire if its first daughter has the value `-` for its `SING` feature and the third daughter has the value `+` for its `PLURAL` feature. Naturally the categories `DET`, `ADJ` and `N` must also match.

Notice that the rule does not imply that the second daughter can not have features; it does not matter what extra features the daughters have so long as the specified ones match. The rule will produce a NP phrase that has the value + for its PLURAL feature, and no other features.

6.1 Feature Matching Semantics

Two values are considered equal for the purposes of matching if they are eq or if one of them is the car of the other, or if one of them is *, or if they are both lists and each subelement matches.

If a category specification specifies a value for a certain feature and the phrase it is matching does not contain a value for that feature (or the value is nil), then the specification matches so long as the feature name does not start with +. So the specification (NP PLURAL -) will match (NP) and (NP PLURAL -), but (NP +PLURAL -) will only match (NP +PLURAL -), not (NP).

6.2 Unification

If a specification contains a symbol starting with ? in a feature value slot, then that symbol will match any value in that slot. Furthermore, if the same symbol is found in another part of the same or a different specification in the same rule, it must match the same value it matched before. The symbol can be used in the left hand side of a rule to define a feature value. The rule

```
((NP CLR ?X) ==> (DET PLURAL ?A) (ADJ COLOR ?X) (N PLURAL ?A))
```

specifies that the value for PLURAL must be the same in the first and third daughters, and that the value for the COLOR feature in the second daughter will be given to the resulting phrase in the CLR feature.

If a feature is not present in a daughter phrase, then unification does not take place on its value. So if the N daughter does not contain a value for the PLURAL feature (or it is nil), then ?A is left free to vary in the DET phraseq. And if COLOR is not specified for ADJ, then the resulting NP will have no feature specification for CLR.

6.3 The Slash Feature

It is possible to include empty categories in a grammar by just putting in rules of the sort (X ==>) but this often leads to gross overgeneration and inefficiency in a complex grammar, especially in the system used here where the initial parsing is done without feature information. A good solution is to automatically augment the rules with information about empty categories. Empty categories are written ((X / X) ==>) and rules are multiplied out to explicitly include empty category information in the slash feature /. The slash feature is automatically propagated from one daughter to its parent, so that a rule like (A ==> B C) generates two additional rules, ((A / ?SLASH) ==> (B / ?SLASH) C), and ((A / ?SLASH) ==> B (C / ?SLASH)).

The slash feature has special feature-passing semantics associated with it. First of all, slash propagation does not occur in rules that have a slash feature specification in their left hand side. So none of the rules ((A / nil) ==> B C) and ((VP / NP) ==> V) and ((VP / NP) ==> V (NP / NP)) will generate any new subrules. This provides a way to ensure that rules do not allow unwanted empty categories. Secondly, the slash feature is only passed up from

one phrase on the right hand side at a time. This is why the rule `(A ==> B C)` did not generate any rule of the form `((A / ?SLASH) ==> (B / ?SLASH) (C / ?SLASH))`. Thirdly, slashes are never propagated from a category specification that already includes the slash feature. So `(A ==> (B / Z) C)` only generates the one additional rule `((A / ?SLASH) ==> (B / Z) (C / ?SLASH))`. Finally, if a specification includes a slash then it will only match a phrase that has a slash feature, *i.e.*, `(X / ?SLASH)` will match `(X / Z)` but not `(X)`.

Slashes are folded into the featureless rules used by the CFG parser in the initial stage of parsing a GPSG grammar. This is why it is more efficient to parse with slashed rules. Instead of the rule `((A / ?SLASH) ==> B (C / ?SLASH))` being represented as `(A ==> B C)` it actually becomes `(A/X ==> B C/X)` and `(A/Y ==> B C/Y)` and so on for all possible empty categories `X` and `Y`. While this greatly multiplies the number of rules it turns out to be significantly more efficient (figure out why for yourself). Many of the rules are removed before parsing by some obvious filters.

7 Function Summary

7.1 Loading and Saving Rule Sets

`(save-rules file-name &rest rule-set-names)`
Save rule sets to a text file. If `rule-set-names` is `nil`, then all possible are saved.

7.2 Manipulating Rule Sets

`(list-all)`
List all the rule sets names, their types, and how many rules are in them.

`(rule-set name &optional type)`
Returns the rule set `name` if it is of type `type` (always if `type`) is not given. A rule set is of the form `(name type rule-list)` where the `rule-list` is the list of rules.

`(add-rule-set name type)`
Create a rule set `name` of the type `type`. Marks the new rule set for editing using `add`, `del`, and `rpl`.

`(remove-rule-set name)`
Remove the rule set `name`.

`(sort-rules name)`
Alphabetically sort the rules in the rule set `name`.

`(list-rules name &optional (template '*))`
List all the rules in the rule set `name` that match the given template.

`(add rule1 rule2 ...)`

Add rules to the last rule set listed with `list-rules` or last rule set created.

`(del n1 n2 ...)`
Remove the numbered rules from the last rule set listed with `list-rules` or last rule set created.

`(rpl n1 rule1 n2 rule2 ...)`
Replace the numbered rules with the new ones in the last rule set listed with `list-rules` or last rule set created.

`(add-rule name rule)`
Add `rule` to the rule set `name`.

`(del-rule name rule)`
Remove `rule` from the rule set `name`.

`(add-rule-list name rule-list)`
Add the rules in the list `rule-list` to the rule set `name`.

`(del-rule-list name rule-list)`
Remove the rules in the list `rule-list` from the rule set `name`.

7.3 Creating Grammar Tables

`(create-cfg-table cfg-table-name cfg-name root k)`
Create the rule set `cfg-table-name` by translating the CFG `cfg-name` into a CFG-TABLE, under the assumption that `root` is the root node. If `k` is 0 no lookahead table is created. If it is 1, one is.

`(create-gpsg-table gpsg-table-name gpsg-name root k)`
Create the rule set `cfg-table-name` by translating the GPSG `cfg-name` into a GPSG-TABLE, under the assumption that `root` is the root node. If `k` is 0 no lookahead table is created. If it is 1, one is. The root should be a symbol, not a featured category.

7.4 Parsing Sentences

```
(p sentence &key (template '*) grammar dictionary (use-lookahead t)
                 print-states file sentence-set)
```
See description in text. If a sentence set (a rule set of type SENTENCES) is specified, then what is returned is a list of elements, the car of which is the sentence and the cdr the list of parses. When parsing a sentence set, the given `sentence` is appended onto the sentence set. If a file name is specified, all output is to that file.

`(retrieve-parses &optional (template '*) &key print-states)`
Return the results of the last sentence parse which match the template `template`. If a GPSG grammar is being used and `print-states` is set true, then unification information is printed during retrieval.

`(states start end &key matching print-rules print-parses)`
Print debugging information on states that completed phrases from `start` to `end`.