SCP: A Simple Chunk Parser

Philip Brooks
Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602-7415 U.S.A.
http://www.ai.uga.edu/

May 8, 2003

1 Introduction

1.1 Chunk Parsing

Chunk parsing, also called shallow parsing or partial parsing, is a technique described by Steven Abney (1991). Intuitively, when one reads a sentence, one deciphers it a piece at a time. A chunk parser attempts to model human parsing by breaking the text up into small pieces, each parsed separately. Chunk boundaries correspond roughly to the pauses in everyday speech.

Abney defines chunks in terms of major heads:

> Major heads are all content words except those that appear between a function word f and the content word that f selects. For example, *proud* is a major head in *a man proud of his son*, but *proud* is not a major head in *the proud man*, because it appears between the function word and the content word *man* selected by *the*. . . . Let h be a major head. The root of the chunk headed by h is the highest node in the parse tree for which h is the s-head, that is, the 'semantic' head. Intuitively, the s-head of a phrase is the most prominent word in the phrase. (p. 2)

By this definition, chunks are non-recursive (never containing a phrase of the same category as itself). It is therefore possible to employ regular expressions to parse them. This approach is less expensive computationally than a full parser (Abney 1996), but it is not the tactic SCP uses.

If the non-recursive grammar constraint is enforced, X-Bar theory is not a valid foundation upon which to build a grammar. An $\bar{\text{X}}$ must be able to contain another $\bar{\text{X}}$ as a constituent, violating the constraint. To get around this limitation, chunk grammars often employ simplex clauses, flat trees in which the specificer, adjuncts, and complement of a phrase are sibling daughters to it. For example, the sample grammar included with SCP defines NP as follows:

NP ⟶ D? AdjP? AdjP? AdjP? N.

where ? indicates the immediately preceding constituent is optional. The result of parsing the NP

the big red balloon

with the previous rule would produce the tree

[NP [D *the*] [AdjP [Adj *big*]] [AdjP [Adj *red*]] [N *balloon*]].

Note that the D, AdjPs, and N are all siblings. This eliminates the need for recursion involving $\bar{X}$s.

Abney's chunks are sub-graphs of the entire tree for a sentence. A natural accompaniment for a chunking parser is an attacher, a parser which takes these sub-graphs and unites them into a tree for the entire sentence. The two programs could be taken together as a two-pass parser.

In addition to perhaps being a better model of human behavior than full parsing methods, other advantages of chunk parsing are as follows:

- Speed. Because a chunk parser only needs to deal with small, non-recursive clauses, it is able to process text much more quickly.
- Small footprint. Smaller phrases also require much less memory to parse. SCP performs cuts after each chunk is parsed, so there are no backtrack points between one phrase and another.
- Robustness. When a full parse fails, it must discard an entire sentence, even if it got much of the structure correct. A chunk parser only discards a few words when it cannot figure out how to proceed.

1.2 The Software

SCP is a simple top-down parser able to handle phrases with optional constituents. It takes a grammar of Prolog clauses and a stream of words as its input, and its output is a stream of chunk structures. It uses a full-blown parsing algorithm instead of regular expressions for flexibility. This is slower than a finite-state parser, but does not bind the user to Abney's approach. For example, it is possible to use recursive rules without trouble.

Given an appropriate grammar, it is possible to use SCP as an attacher, something regular expression parsers cannot do because of their inability to handle recursive structures. SCP can also refer to an external lexicon such as WordNet (Princeton 2003) instead of a lexicon internal to the grammar. Finally, SCP includes predicates (most notably `flatten_chunks/2`) to extract the text (and optionally the category) from a chunk, which may be interesting to those wishing to study the similarities between chunks and prosodic patterns of speech.

SCP is loosely based on the top-down parser on page 154 of Covington (1994a) and materials from his Natural Language Processing class in Spring 2003.

[1]So long as they are not left-recursive, where a phrase's first constituent is of the same type as itself. In these cases the parser enters an infinite loop. This is a limitation common to the top-down parsing algorithm.

2 The Grammar

SCP uses user-defined Prolog predicates `word/2` and `rule/2` to represent its grammar. Each such predicate describes a single word or rule. The first argument of either is the category of the word or phrase being described. The second argument for `word/2` is a Prolog term denoting the word. The second argument of `rule/2` is a list. Each of the members of the list is a constituent of the phrase, in the order given. Brackets around a single constituent indicate that it is optional. For example, the rule

```
rule( np,    [ [d] [adjp] [adjp] [adjp] n ] ).
```

would match *the big red balloon* or *dog* but not *a tiny colorless* (because it lacks a noun) or *democratic a government* (because it is in a different order).

The core of an SCP grammar is the `chunk` meta-phrase. When the chunk parsing predicates such as `chunk_parse/3` are called, they attempt to pull the first phrase of category `chunk` from the input stream. You must specify what phrases are to be considered chunks. In the sample grammar, the phrases are NP, PP, and VP, among others. The rules which specify them as chunks are:

```
rule( chunk,    [ np ] ).

rule( chunk,    [ vp ] ).

rule( chunk,    [ pp ] ).
```

It is also a good idea to have unknown words and phrases marked as chunks at the end of your grammar, like so:

```
rule( chunk,    [ unrecognized_word ] ).

word( unrecognized_word, _ ).
```

Otherwise, SCP may fail when it tries to parse a word not in its lexicon.

If you do not wish to use an internal lexicon, you can access an external one by writing a `word/2` clause which calls the interface to the dictionary. For example, if you wish to refer to WordNet 1.6 or 1.7.1, a crude interface which uses WordNet's numbers instead of atoms for categories could be implemented as follows:

```
word(Category,Word) :-
       s(_,_,Word,Category,_,_).
```

A sample grammar is included in the SCP distribution, in the file `grammar.gmr`. To use it, simply consult it. There are four small texts included, stored in the predicate `text/2`. Call the predicate `test(+Number)` to watch SCP in action using the sample texts, where `Number` corresponds to the number of the sample text you want to parse.

[2]Assume that the other words and phrases mentioned are defined appropriately.

3 Tree Formation

When SCP matches a word while parsing, it creates a unary (single argument) Prolog structure with the category of the word as its functor and the word itself as the argument. This forms a two-node subtree called a word-structure. For example, suppose SCP is looking for a noun and must match the word *meeting*. It looks into its grammar and finds the entry

```
word( n, meeting ).
```

The match succeeds, and the word-structure `n(meeting)` is formed.

Rules are a bit more complicated, but processed similarly. The right-hand side of a rule can match any number of constituents. These constituents can themselves be either categories of words or of other phrases. If the match is successful, SCP creates a phrase-structure, an n-ary Prolog structure where n is the number of constituents the rule matched (excluding any optional constituents which did not match). The xth argument of the phrase-structure is the word-structure of the xth matched constituent (if it is a word) or the phrase-structure of the xth matched constituent (if it is a phrase).

For example, parsing *the old grey mare* for a chunk with the grammar

```
rule( chunk,    [ np ] ).

rule( np,       [ [d], [adjp], [adjp], [adjp], n ] ).

word( d, the ).
word( n, mare ).

rule( adjp,     [ adj ] ).

word( adj, old ).
word( adj, grey ).
```

gives you the phrase-structure

```
chunk(
      np(
         d(the)
         adjp(
              adj(old)
             )
         adjp(
              adj(grey)
             )
         n(mare)
        )
     ).
```

[3]Or the same phrase, if you choose to allow recursion in your grammar.

4 Limitations

Because SCP is not a regular expression grammar, it is not as fast as a chunk parser potentially could be. It also does not support operations such as the Kleene *, which matches nothing or any number of occurrences of a pattern.

It also does not support GULP (Covington 1994b) for implementing a unification-based grammar. This is not a major limitation for chunk parsing because in the interest of speed, feature matching is often not employed. It does pose an inconvenience to one wishing to use SCP as an attacher.

5 Predicates

This section documents the public (user-accessible) predicates SCP provides.

5.1 Parsing Predicates

chunk_parse(+Stream, −Rest, −Tree)

This predicate takes `Stream`, which must be a list of words, and parses the first chunk from it. `Tree` is instantiated to the tree of the chunk, and `Rest` is the rest of the stream once the chunk is removed.

chunk_parse_text(+Stream)

This predicate calls `chunk_parse/3` until the stream of words `Stream` is exhausted. It then displays the results as indented structures using `show_tree/1`. Here is one of the sample texts from `sample.gmr` run through as an example (this is equivalent to calling `test(4)`):

```
?- chunk_parse_text([this, is, max, smith, '.']).
chunk
   np
      pronoun
         this

chunk
   vp
      vaux
         is

chunk
   np
      name
         first_name
            max
         last_name
```

```
            smith

chunk
   separator
      unspoken_separator
```

chunk_parse_text(+Stream, −List)

As `chunk_parse_text/2` above, except instead of writing the stream of chunks, it fills `List` with them. Here's another sample text, run through this predicate:

```
?- chunk_parse_text([this,is,the,cat,
                     that,chased,the,rat,
                     that,lived,in,the,house,
                     that,jack,built],X).

X = [chunk(np(pronoun(this))), chunk(vp(vaux(is))), chunk(np(d(the),
n(cat))), chunk(separator(spoken_separator(that))), chunk(vp(v(chased))),
chunk(np(d(the), n(rat))), chunk(separator(spoken_separator(that))),
chunk(vp(v(lived))), chunk(pp(p(in), np(d(the), n(house)))),
chunk(separator(spoken_separator(that))), chunk(np(name(first_name(jack)))),
chunk(vp(v(built)))]
```

parse(?Category, +Stream, ?Rest, ?Tree)

This is the heart of the parser. It requires a stream of words `Stream` from the beginning of which it parses a phrase or word of the category `Category` with `Rest` remaining. `Tree` is unified with the tree of the parsed graph. All the other parsing predicates call this one.

`parse/4` can be used for more than just parsing whole chunks. For example, to pull the initial NP off *the big dog lived* using `sample.gmr`, do the following:

```
?- parse(np, [the,big,dog,lived], Rest, Tree).

Rest = [lived]
Tree = np(d(the), adjp(adj(big)), n(dog))
```

5.2 Miscellaneous Predicates

show_tree(+Tree)

`show_tree/1` takes a graph `Tree` as its argument and writes it one node per line, with indentation indicating dominance. For example:

```
?- show_tree(chunk(np(d(the),n(dog)))).
chunk
   np
```

```
d
   the
n
   dog
```

flatten_chunks(+Chunks,−List)

`Chunks` must be instantiated to a list of chunk trees. The text from each chunk is extracted into a list. Each of these is placed into the list `List` in order. An example:

```
?- flatten_chunks([ chunk(np(d(the),n(dog))),
                    chunk(vp(v(ate))),
                    chunk(np(d(the),n(shoe)))
                  ], X).

X = [[the, dog], [ate], [the, shoe]] ;
```

flatten_chunk(+Chunk,−List)

This predicate takes `Chunk` to be the tree of a chunk and extracts the words from it, instantiating `List` to them. It calls `flatten_chunk/3` and discards the third argument. An example:

```
?- flatten_chunk(chunk(pp(p(to),np(d(the),n(batcave)))),X).

X = [to, the, batcave]
```

flatten_chunk(+Chunk,−List,−Type)

`flatten_chunk/3` takes a chunk tree `Chunk` and instantiates `Type` to the category of the chunk (syntactically, the functor of `X` in `chunk(X)`) and `List` to the list of the words in the chunk. For example:

```
?- flatten_chunk(chunk(np(d(the),n(cat))),List,Type).

List = [the, cat]
Type = np
```

References

[1] Abney, Steven (1991) Parsing by Chunks.
http://www.vinartus.net/spa/publications.html.

[2] Abney, Steven (1996) Partial Parsing via Finite-State Cascades.
http://www.vinartus.net/spa/publications.html.

[3] Covington, Michael (1994a) *Natural Language Processing for Prolog Programmers*. Prentice Hall: 1994.

[4] Covington, Michael (1994b) GULP 3.1: An Extension of Prolog for Unification-Based Grammar. Research Report AI-1994-06, Artificial Intelligence Center, The University of Georgia.

[5] Princeton (2003) WordNet 1.7.1. http://www.cogsci.princeton.edu/∼wn/.