Parsing Bounded Discontinuous Constituents: Generalisations of some common algorithms

Mike Reape

Centre for Cognitive Science and
Department of Artificial Intelligence
University of Edinburgh*

email: reape@cogsci.ed.ac.uk

Abstract

This paper presents generalizations of several common parsing algorithms for parsing languages with *bounded* discontinuous constituency. An examination of the original algorithms shows how they maintain the three crucial properties of *minimality*, *soundness* and *completeness*. Respectively, these properties require that a parsing algorithm find a given analysis at most one time, that every analysis is an analysis defined by the grammar and that, if the language is decideable, every analysis will be found. This allows us to explain straightforwardly how the generalised algorithms extend the original ones. We also consider the computational complexity of the classes of grammars parsed by these algorithms briefly.

1 Introduction

An outstanding characteristic of modern grammatical theory is that it has become increasing lexicalist, i.e., that most grammatical information is encoded in the lexicon and that the syntactic components of grammars have become increasingly impoverished. This is true to varying degrees of Government and Binding Theory (GB) and Lexical Functional Grammar (LFG) and is particularly true of the development of first Head Grammar (HG) and then Head-driven Phrase Structure Grammar (HPSG) from Generalized Phrase Structure Grammar (GPSG).

In HPSG, this has been carried to the point where all syntactic and phonological information is encoded in *signs*. Syntactic structure is encoded as a complex feature-structure which is the value of a DAUGHTERS (DTRS) attribute and not as a tree. The PHONOLOGY (PHON) attribute of a sign is a sequence of atoms representing the phonology (or orthography) of the entire sign. The mapping from the DTRS attribute to the PHON attribute is determined by a relation called *order-constituents*.[1] The following implication states the *relational dependency* of the value of the PHONOLOGY attribute on the value of the DTRS attribute in phrasal signs.

*Formerly also of Vakgroep α-informatica, Universiteit van Amsterdam.

[1] Although Pollard and Sag ([14]; henceforth P&S) state "the **phonology** value of a phrasal sign is specified as a function, called *order-constituents*, of the sign's **daughters** value", order-constituents is really a relation since a given set of daughters may give rise to more than one possible word ordering in languages with semi-free word order.

$$\textit{phrasal-sign}[\ ] \Rightarrow \begin{bmatrix} \text{PHON} & \text{order-constituents}(\boxed{1}) \\ \text{DTRS} & \boxed{1} \end{bmatrix}$$

This implication is taken to be a "language specific principle of grammar". That is, the relational behaviour of order-constituents is language specific. What is of fundamental importance here is that this relation cannot order the phonology of a sign in terms of daughters since there is in general no linearly ordered encoding of all the daughters of a phrasal sign.

For example, in a head-complement sign, DTRS takes as its value a feature structure containing a HEAD-DAUGHTER feature and a COMPLEMENT-DAUGHTERS feature which is a sequence of complement daughter signs.

$$\begin{bmatrix} \text{DTRS} & \begin{bmatrix} \text{HEAD-DTR} & [\ldots] \\ \text{COMP-DTRS} & \langle \ldots \rangle \end{bmatrix} \end{bmatrix}$$

However, in a filler-gap sign, DTRS takes as its value a feature structure containing a FILLER-DAUGHTER feature and a HEAD-DAUGHTER feature.

$$\begin{bmatrix} \text{DTRS} & \begin{bmatrix} \text{FILLER-DTR} & [\ldots] \\ \text{HEAD-DTR} & [\ldots] \end{bmatrix} \end{bmatrix}$$

Therefore it is impossible to derive the order of the PHON value by concatenating the phonologies of the daughters from left to right (where *all* the daughters are ordered). This leaves us with the situation that signs determine word order directly instead of word order being determined by the "leaves" of surface syntax trees as in standard syntactic-based approaches to word order.

This makes a great deal of freedom available for determining word order. Among other things, it means that it is possible to define the mapping from syntactic structure to phonology to allow *bounded* discontinuity. By bounded discontinuity we mean phenomena such as Mittelfeld "scrambling" in German and so-called "cross-serial" dependencies in Dutch as opposed to *unbounded* discontinuity such as *Wh*-movement.

For example, it is possible to define the mapping such that the phonologies of two daughter constituents get interleaved in the phonology of the mother. Consider the following example from P&S (ex. 373, p. 189).

$$\begin{bmatrix} \text{PHON} & \langle \text{the walked poor to old the man store} \rangle \\ \text{SYN|LOC} & \begin{bmatrix} \text{HEAD} & [\text{MAJ V}] \\ \text{SUBCAT} & \langle\ \rangle \end{bmatrix} \\ \text{DTRS} & \begin{bmatrix} \text{HEAD-DTR} & \begin{bmatrix} \text{PHON} & \langle \text{walked to the store} \rangle \\ \ldots \end{bmatrix} \\ \text{COMP-DTRS} & \left\langle \begin{bmatrix} \text{PHON} & \langle \text{the old man} \rangle \\ \ldots \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix}$$

Notice that in this example, constituency boundaries place no constraints whatsoever on the possibility of of "interleaving" the phonologies of daughters (and permuting them arbitrarily)

to produce the phonology of the "mother". Such a state of affairs characterises the so-called W^* languages. Of course, there is hardly a linguist to be found who really believes in the existence of such languages. However, there has been a steady progression of linguists who have proposed "string combination/composition operators" stronger than simple concatenation to provide solutions for apparent cases of discontinuity (i.e., nonadjacency of subconstituents) rather than appealing to "noncanonical" syntactic representations.[2] A short list of these include Bach's *left- and right-wrap operators* ([2, 3]), Pollard's *head-wrapping operators* ([13]), the *order-constituents* relation of HPSG ([14]), Dowty's *ordering* and *attachment* operators ([6]), Reape's *domain union* operator ([16, 17, 18]) and the cb (*combine*) predicates of van Noord ([11]).

One way to conceive of the use of these operators is in terms of derivation trees. Given a local tree

$$[_M D_1, ..., D_n]$$

(for $n > 1$) each daughter D_i will be annotated with the string it derives. Associated with the mother node M will be a string combination operator $f_M(x_1, \ldots x_n)$. M will then be annotated with $f_M(\pi(D_1), \ldots, \pi(D_n))$ (where $\pi(D_i)$ is the string that D_i is annotated with for each i). That is local trees will be pairs of (X, π) where X is a syntactic category and π is the string it derives. To put it informally, each node "knows" what string it derives.

Since we are bringing syntactic categories and their derived string (at least) together into a unit, I will henceforth call these units *signs* following the practice of both HPSG and UCG. Thus, the word "category" will refer unambiguously to syntactic category and not any larger unit of linguistic structure. However, in what follows, I will typically not include either phonological (PHON) or derivational (DTRS) information (attributes) in the sign since we will not be considering any particular set of string combination operators. Rather our goal, will be to investigate parsing algorithms which can be used with any grammar which constructs derived strings within the *signs* themselves.

Given that a grammar of this type determines word order in the sign, the question that arises is what the parser should do. The answer is very simple. Given a grammar G with a set of productions of the form $M \longrightarrow D_1, \ldots, D_n$, for each production, assign an arbitrary order to any daughters which are not already ordered. Call the new grammar formed G'. Then consider the language $L(G')$ generated by G'. Next, define the language $L^{perm} = \{s' | s \in L(G') \text{ and } permutation(s, s')\}$ where $permutation(s, s')$ is true iff s' is a permutation of s. L^{perm} is the *permutation closure of* L. We say L^{perm} is *permutation-complete* or *permutation-closed.* This means that for an input string of length n, we have to consider every permutation of the string when parsing, i.e., we have to consider $n!$ permutations and the parsing problem is for *permutation-complete* languages.

One might object at this point that a particular grammar will never actually have a search space that large and that the algorithms should be specialised to build the string-combining operations directly into the parser. There are two responses to such criticism. First, we have set out to find parsing algorithms for a class of grammars whose nonterminal symbols determine word order. Therefore, it is the function of the grammar to determine order and not the parsing algorithms. Furthermore, as such theories of word order evolve, the mechanisms for detemining word order are likely to change. Thus, it is premature to "fold" theories of word order and their string-combination operations into the parser.

Second, the primary goal of this work is to identify what changes need to be made to a class of algorithms to make them parse permuatation-complete languages while maintaining the properties

[2]The term *operators* strictly suggests that they are in fact functions. In §8 we will consider an "operator" which is only relational.

of *minimality*, *soundness* and *completeness* (assuming that the original algorithms have those properties). Minimality requires that any given derivation is only produced once. Soundness requires that the parser only accepts strings which are in the language generated by the grammar. Completeness requires that the parser produces every derivation of a string that is generated by the grammar.

We will now present several common parsing algorithms all of which are minimal, sound and complete and show how they can be generalised to parse permutation-complete languages while maintaining the three properties described. In what follows, the parsers will be presented as Prolog programs. We will sometimes make use of the Definite Clause Grammar (DCG) notation to simplify the presentation. In each case, we will present a Prolog version of a common algorithm and then present its generalisation along with an informal indication of why the generalised parser is minimal, sound and complete. We will also assume a uniform rule format of the form `rule(Mom,Kids)` where `Mom` is the mother sign and `Kids` is a Prolog list of daughter signs. (For the head-corner parser, rules take the form `rule(Head,Mom,Kids)` where `Head` is the head daughter.) Lexical entries are of the form `lexicon(Sign,Word)` where `Sign` is the sign of the word `Word`. Lexical entries could be easily generalized to allow lists of words but we won't consider this refinement here. No particular assumptions are made about the structure of signs except that they are Prolog unifiable terms. This restriction can be easily eliminated by making the call to a specialized unification predicate explicit but we won't bother to here.

2 A generalised top-down parser

A simple top-down parser which uses the Prolog control strategy can be defined with just three predicates for the type of grammars that we consider. `parse/3` takes as input a (possibly uninstantiated or partially instantiated) sign, a list of words and returns a suffix of the list. (We will assume henceforth that this is always instantiated to `[]`.) The first clause finds a rule the head of which the goal sign `Mom` unifies with and finds the list of daughter categories `Kids`.

```
parse(Mom) -->
  {rule(Mom,Kids)},
  parse_kids(Kids).
```

The daughters are then parsed one after the other by `parse_kids/3`.

```
parse_kids([]) --> [].
parse_kids([Kid|Kids]) -->
  parse(Kid),
  parse_kids(Kids).
```

The second clause for `parse` calls `connects/3` to take the first word `Word` off the input string and return the tail. The sign `Sign` which is the input to `parse` and `Word` are passed to `lexicon/2` to see if there is a lexical entry for `Word` of sign `Sign`.

```
parse(Sign) -->
  connects(Word),
  {lexicon(Sign,Word)}.

connects(Word, [Word|Rest], Rest).
```

This completes the description of the basic top-down parser. The major point to note is that `connects` forces a strict left to right discipline for unifying lexical entries with signs. To generalise this algorithm to a permutation-complete one we must allow the possibility that *any* word in the input string can be unified with the current sign. We do this by changing the second clause to `parse` so that it calls a predicate `delete` instead of `connect`.

```
parse(Cat) -->
  delete(Word),
  {lexicon(Cat,Word)}.
```

`delete/3` nondeterministically deletes an element from its input list.

```
delete(X,[X|L],L).
delete(X,[Y|L1],[Y|L2]) :-
  delete(X,L1,L2).
```

Consider the following trivial grammar. The categories v and n subcategorise for the arguments in the list in the second argument position. `sign(n,[],NSem)` is an NP, `sign(v,[],VSem)` is an S, `sign(v,[sign(n,[],NSem)],VSem)` is a VP and single common nouns subcategorise for determiners as in HPSG, `sign(n,[sign(det,[],DSem)],NSem)`. The first rule is like 'Rule 1' of P&S. It combines a head with one argument with that argument to give a mother with no arguments. The second rule is like 'Rule 2' of P&S. It combines all of the arguments of a verb except the first NP argument with the verb to produce a VP.

```
% lexicon(Word,Sign)

lexicon(john,sign(n,[],john)).
lexicon(mary,sign(n,[],mary)).
lexicon(boy,sign(n,[sign(det,[],Det)],boy(Det))).
lexicon(dog,sign(n,[sign(det,[],Det)],dog(Det))).

lexicon(kicked,sign(v,[sign(n,[],X),sign(n,[],Y)],kicked(X,Y))).

lexicon(the,sign(det,[],the)).
lexicon(his,sign(det,[],his)).

% rule(Mother,Daughters)

rule(sign(Cat,[],Sem),                          % 'Rule 1'
    [sign(Cat,[X],Sem),X]).
rule(sign(v,[sign(n,[],NSem)],Sem),             % 'Rule 2'
    [sign(v,[sign(n,[],NSem),X|Y],Sem),X|Y]).
```

Notice that this grammar will produce two analyses of "john loves mary", one where John loves Mary and one where Mary loves John.

```
| ?- parse(X,[john,mary,loves],[]).
```

```
X = sign(v,[],loves(john,mary)) ? ;

X = sign(v,[],loves(mary,john)) ? ;

no
```

The parser will also assign these two analyses to the 3! = 6 different permutations of `[john,loves,mary]`.

```
[john,loves,mary]
[john,mary,loves]
[loves,john,mary]
[loves,mary,john]
[mary,john,loves]
[mary,loves,john]
```

Notice also that the sentence "john loves john" receives two analyses, one where the first `john` is the first argument of "loves" and the second the second and one where the first `john` is the second argument and the second the first.

```
| ?- parse(X,[john,loves,john],[]).

X = sign(v,[],loves(john,john)) ? ;

X = sign(v,[],loves(john,john)) ? ;

no
```

This does not mean that the parser is not minimal however. Different occurrences of `john` are being assigned to different argument roles. It is easy to see that the generalised parser is sound, complete and minimal since it considers all permutations of the input string and is identical to the usual top-down parser (which is minimal, sound and complete) in all other details.

3 A generalised left-corner parser

In this section, I'll present a left-corner parser. The presentation is very similar to that in [12] and I will only briefly describe its workings here. First, we define the notion of *left-corner*. *left-corner*(X, Y) is true iff $X = Y$ or there exists a rule $Z \longrightarrow X, D_1, \ldots, D_n$ and *left-corner*(Z, X). The left-corner parser is a bottom-up parser which uses the left-corner relation to determine which rules to try. This helps limit the search space compared to a naive bottom-up parser which traverses the entire search space.

`parse/3` is similar to the same predicate in the top-down parser. It removes the first word from the input list, finds the sign of the word `WSign` in the lexicon and then calls the predicate `left_corner/2` with `WSign` and the goal sign `Sign`. `connect/3` is defined as for the top-down parsers.

```
parse(Sign) -->
  connect(Word),
  { lexicon(Word,WSign) },
  left_corner(WSign,Sign).
```

`left_corner/4` has two clauses. The first says that a sign is a left-corner of itself. The second says that a daughter `Kid` is a left-corner of an ancestor node `Top` if the daughter is the first daughter in a rule with mother sign `Mom`, the rest of the daughters can also be parsed and the mother sign `Mom` is the left-corner of the ancestor node `Top`. This implements the abstract definition of *left-corner*(X, Y) above precisely.

```
left_corner(Sign,Sign) --> [].
left_corner(Kid,Top) -->
  { rule(Mom,[Kid|Kids]) },
  parse_kids(Kids),
  left_corner(Mom,Top).
```

`parse_kids/3` is defined exactly as for the top-down parsers.

As with the top-down parser, generalisation is trivial since the input string "drives" both algorithms. Therefore, we just need to replace the goal `connect(Word)` with the goal `delete(Word)` in `parse/3` precisely as we did in the top-down case.

```
parse(Sign) -->
  delete(Word),
  { lexicon(Word,WSign) },
  left_corner(WSign,Sign).
```

`delete/3` is defined exactly as for the generalised top-down parser. The generalised algorithm is minimal, sound and complete for the same reasons that the generalised top-down parser is.

4 A head-corner parser

In [11], Gertjan van Noord presents a "head-corner parser". Abstracting from the details of his presentation a bit, the head-corner parser can be seen to be a variant of the generalised left-corner parser.[3] van Noord considers a class of grammars which have the *semantic head property*. This means that the semantics of every constituent is a projection of a lexical head called the *seed*. The notion of the seed is defined inductively. The seed of a tree is the seed of its head. The seed of a lexical head is the head itself. Grammatical theories like HPSG and UCG also have the semantic head property. This has also been described in the context of generation algorithms in [5], [9],[10],[20] and [21]. Typically, grammars with the semantic head property also project some lexically specified features from head daughters to mothers. This takes the form of the Head Feature Principle in GPSG and HPSG. This means that there will be lexical feature sharing between the root of a tree and its seed. van Noord's intuition is that the combination of shared syntactic and semantic information should be effective in cutting the lexical search space in parsers for discontinuous constituency, and by implication, permutation closed languages.

The necessary changes occur in two predicates: `parse` and the second clause of `head_corner`, which replaces `left_corner`. A predicate `head/2` is defined which is grammar specific. It is the equivalent of the Head Feature Principle. It projects some of the syntactic and semantic information from the goal sign onto a (partially specified) lexical seed. `parse` takes the goal

[3] The algorithm is similar to but more general than the "head-driven parsing algorithm" of [8]. It is unclear what relationship the algorithm bears to the "head-driven parser" presented in [15].

sign Sign and projects some of its information onto the sign WSign.[4] Then lexicon is called nondeterministically with WSign to hypothesize possible lexical heads. This returns a word Word which delete tries to delete nondeterministically from the input string.[5] Notice that in the left-corner parser Word is uninstantiated when delete is called but in the head-corner parser it is instantiated.[6]

```
parse(Sign) -->
  { head(Sign,WSign),
    lexicon(Word,WSign) },
  delete(Word),
  head_corner(WSign,Sign).
```

In the second clause of head_corner, the head daughter Head replaces the (arbitrary) daughter Kid in the left-corner parser. Rules are of the form rule(Head,Mom,Kids) where Head is the head daughter, Kids are the other daughters and Mom is the mother sign. As in the left-corner parsers, the first argument (Head) is required to be one of the daughters of the rule and then the rest of the daughters Kids are parsed by parse_kids. Finally, the mother sign Mom and the goal sign Top are passed to head_corner analogously to the left-corner parsers.

```
head_corner(Sign,Sign) --> [].
head_corner(Head,Top) -->
  { rule(Head,Mom,Kids) },
  parse_kids(Kids),
  head_corner(Mom,Top).
```

Given the soundness of the head relation with respect to the grammar and the fact that the generalised left-corner parser is minimal, sound and complete, it is easy to see that the head-corner parser is also minimal, sound and complete.

The predicate head is very simple. It just unifies the major category symbol and semantics of the goal and the seed.

---

[4] Of course, if the goal sign is the totally unspecified sign, lexical search will not be reduced at all.

[5] In van Noord's version, delete/3 is replaced by the predicate subset/3 to allow for complex lexical heads which have discontinuous components, e.g., Dutch and German separable prefix verbs and English particle-verbs. subset and its subsidiary predicate select_chk are defined as follows.

```
subset([],P,P).
subset([H|T],P0,P) :-
  select_chk(H,P0,P1),
  subset(T,P1,P).

select_chk(E1,[E1|P],P) :- !.
select_chk(E1,[H|P0],[H|P]) :-
  select_chk(E1,P0,P).
```

The cut in the first clause of select_chk can be eliminated if one wants to allow two analyses for sentences like "john loves john".

[6] The nondeterministic lexical access before deletion from the input string seems like it might be inefficient if the number of lexical heads projected from the goal sign is potentially very large. van Noord's implementation includes the obvious remedy to this problem. Lexical access is restricted to just those words which occur in the input string. Notice that the calls to lexicon and delete are in the reverse order to the calls to connect and lexicon in the left-corner parser. It is precisely this reversal which gives rise to the possible reduction in the search space.

```
% head(Goal,Seed)

head(sign(Syn,_,Sem),sign(Syn,_,Sem)).
```

Now consider the previous grammar but with the rules `rule/2` replaced by rules of the form `rule(Head,Mom,Kids)` where `Head` is the head daughter of the corresponding rule of arity 2.

```
% rule(Head,Mother,Daughters)

rule(sign(Cat,[X],Sem),
     sign(Cat,[],Sem),
     [X]).
rule(sign(v,[sign(n,[],NSem),X|Y]],Sem),
     sign(v,[sign(n,[],NSem)],Sem),
     [X|Y]).
```

With this grammar the following examples can be parsed yielding four analyses each corresponding to *his dog kicked the boy*, *the dog kicked his boy*, *his boy kicked the dog* and *the boy kicked his dog*.

```
| ?- parse(X,[the,boy,kicked,his,dog],[]).

X = sign(v,[],kicked(dog(his),boy(the))) ? ;

X = sign(v,[],kicked(dog(the),boy(his))) ? ;

X = sign(v,[],kicked(boy(his),dog(the))) ? ;

X = sign(v,[],kicked(boy(the),dog(his))) ? ;

no
| ?- parse(X,[the,his,kicked,boy,dog],[]).

X = sign(v,[],kicked(dog(his),boy(the))) ? ;

X = sign(v,[],kicked(dog(the),boy(his))) ? ;

X = sign(v,[],kicked(boy(his),dog(the))) ? ;

X = sign(v,[],kicked(boy(the),dog(his))) ? ;

no
```

5 A generalised shift-reduce parser

In this section, we present a standard shift-reduce parser. Shift-reduce parsers have been advocated by Shieber, Tomita and Pereira among many others. [1] (Chap. 5) provide a thorough introduction to a variety of shift-reduce algorithms. The one we present here should be very

familiar to a computational linguistics audience. The following description will therefore be fairly brief.

The shift-reduce parser is a nontabular, bottom-up parser which uses a stack as its main data structure. When a constituent has been parsed it is added to the top of the stack. At this point there are two choices. Either the stack is left unaltered and further constituents are parsed bottom-up (a "shift") or else a rule is chosen nondeterministically and there is an attempt to unify the n daughters of the rule in reverse order against the top n elements of the stack. If the unification is successful, then the top n elements of the stack are popped and the mother is pushed on the stack. This is called a "reduce" step. The key here is that the stack elements are in reverse order to the way they were parsed. Therefore, their terminal strings conform to the input list.

Therefore, there are two major predicates: `parse/3` which takes an input list of words to be parsed, an input stack and an output stack. The input list is successfully parsed when a single sign is left on the stack and the entire input list has been consumed.

`parse/2` is simply used to initialize the stack to be empty in `parse/3`.

```
parse(L,X) :-
  parse(L,[],X).
```

`parse/3` has two clauses. The first simply returns a single element stack if the input string is empty. This is the termination condition mentioned above. The second clause is tail-recursive on the input list. It takes the first word off the input list, finds the word's sign in the lexicon, passes it to the predicate `reduce` along with the stack and gets a new stack back. This is then passed tail-recursively to `parse` with the rest of the input list.

```
parse([],[X],[X]).
parse([Word|Words], Stack0, Stack) :-
  lexicon(Word, Sign),
  reduce(Sign, Stack0, Stack1),
  parse(Words, Stack1, Stack).
```

The predicate `reduce/3` implements the "shift" and "reduce" operations. The first clause simply "shifts" the sign onto the stack. The second attempts to find a rule with n daughters which which unify with the top n elements of the stack (in reverse order). This yields a new stack which is then recursively passed to `reduce` along with the mother sign `Mom`. If no additional reductions can be performed, then the shift clause succeeds and control returns to `parse`.

```
reduce(Kid, Stack, [Kid|Stack]).          % shift
reduce(Kid, Stack0, Stack) :-             % reduce
  rule(Mom, Kids),
  find_kids(Kids, [Kid|Stack0], Stack1),
  reduce(Mom, Stack1, Stack).
```

The predicate `find_kids` unifies the daughters of a rule against the top of the stack in reverse order, deleting the daughters from the stack.

```
find_kids(Kids, Stack0, Stack) :-
  reverse(Kids, RevKids),
  append(RevKids, Stack, Stack0).
```

We'll now present a generalised version of the shift-reduce parser described above for permutation-closed languages.[7] The first important point about the generalised parser is that it replaces the stack with a multiset (which we will represent as a Prolog list). Thus, the parser might be called a "multiset parser". `parse/2` is defined exactly as for the previous parser. `parse/3` is also defined exactly as for the previous parser except that we rename any variables `Stack`n by `Multiset`n.

```
parse([],[X],[X]).
parse([Word|Words], Multiset0, Multiset) :-
  lexicon(Word, Sign),
  reduce(Sign, Multiset0, Multiset1),
  parse(Words, Multiset1, Multiset).
```

The first clause of `reduce` remains the same, i.e., it "shifts" the sign into the multiset. The second clause is slightly different. As before a rule is found with mother `Mom` and daughters `Kids`. However, instead of shifting `Kid` into the multiset prior to finding elements of the multiset which unify with the daughters of the rule, a call to `delete/3` requires that `Kid` be one of the daughters `Kids` of the rule. The rest of the daughters `KidsExceptKid` are removed from the multiset by `subset/3` which returns the remaining multiset elements. This replaces `find_kids` in the shift-reduce parser. Finally, `reduce` is called recursively with the mother sign `Mom` and the multiset analogously to the shift-reduce parser.

```
reduce(Kid, Multiset, [Kid|Multiset]).          % shift
reduce(Kid, Multiset0, Multiset) :-             % reduce
  rule(Mom, Kids),
  delete(Kid,Kids,KidsExceptKid),
  subset(KidsExceptKid, Multiset0, Multiset1),
  reduce(Mom, Multiset1, Multiset).
```

`subset/3` is defined in the usual way in terms of `delete/3` (without a cut).

```
subset([], Multiset, Multiset).
subset([H|T], Multiset0, Multiset) :-
  delete(H, Multiset0, Multiset1),
  subset(T, Multiset1, Multiset).
```

Now reconsider the grammar of §2. If the following goal is given to the Prolog interpreter,

```
| ?- parse([the,boy,kicked,his,dog],X), write(X), nl, fail.
```

then the following four analyses are produced:

```
[sign(v,[],kicked(boy(his),dog(the)))]
[sign(v,[],kicked(dog(the),boy(his)))]
[sign(v,[],kicked(boy(the),dog(his)))]
[sign(v,[],kicked(dog(his),boy(the)))]
```

[7] The generalised algorithm was developed jointly with Evelyn van de Veen and Pete Whitelock. A version of this algorithm is described in [22].

The second clause of reduce would be exactly analogous to the corresponding clause of the shift-reduce parser if it was of the form:

```
reduce(Kid, Multiset0, Multiset) :-
  rule(Mom, Kids),
  subset(Kids, [Kid|Multiset0], Multiset1),
  reduce(Mom, Multiset1, Multiset).
```

However, in this case there is no guarantee that Kid becomes one of the daughters in the rule application. This destroys the minimality of the algorithm. It is fairly easy to see why. Imagine an input list [a,b,c,d,e] and rules and lexical entries:

```
rule(ab, [xa, xb]).
rule(abc, [xc, ab]).
rule(abcd, [xd, abc]).
rule(abcde, [xe, abcd]).

lexicon(a, xa).
lexicon(b, xb).
lexicon(c, xc).
lexicon(d, xd).
lexicon(e, xe).
```

Since the shift clause of the algorithm will always succeed first, the signs of all the lexical entries in the input string will be shifted into the multiset so we will end up with the multiset [xe,xd,xc,xb,xa]. The ab rule can then be applied and a reduce step performed. The other three rules can also be applied giving three more reduce steps leading to an eventual parse of the input string as category abcde. However, on backtracking, the last lexical category shifted into the multistack, namely xe, will eventually be removed and a reduce step will be attempted. But then the ab rule can still apply. We can also remove xd and apply the ab rule and then remove xc and apply the ab rule. So the non-minimality comes from the interaction of the subset and reduce predicates and the rules. For an input string of length n, we will always have a choice of shifting every lexical entry in the input string and then reducing, shifting the first m lexical entries (for each $0 < m < n$) and then reducing or just shifting exactly what is required for the various rules and then reducing. The resulting combinatorial blow-up will occur even with linguistically motivated grammars. Notice as well, that the number of times that a particular "analysis" is found will not be constant across the different analyses. This is a complex factor of the order of the words in the input string and the rules in the grammar.

So, what is it about the following definition of reduce which makes it minimal?

```
reduce(Kid, Multiset, [Kid|Multiset]).          % shift
reduce(Kid, Multiset0, Multiset) :-             % reduce
  rule(Mom, Kids),
  delete(Kid,Kids,KidsExceptKid),
  subset(KidsExceptKid, Multiset0, Multiset1),
  reduce(Mom, Multiset1, Multiset).
```

The answer is that Kid (which is either a new lexical entry or the sign produced by the last reduce step, i.e., a mother sign) is required to be a daughter in the second clause of reduce by

the call to `delete`. This means that `Kid` is combined with subsets of the multiset in every possible way licensed by the rules but each such combination occurs only once. The problem with the non-minimal algorithm is that the second clause effectively acts as a shift *and* a reduce step.

The generalised algorithm is sound and complete for the same reasons that the standard shift-reduce algorithm is. To elaborate slightly, the algorithm is complete precisely because the reduce step tries to use the last sign produced as a daughter of every rule.

This characteristic of the generalised version of `reduce` is very important because it indicates why the standard shift-reduce parser is minimal and complete. In the standard parser, `Kid` is put on the top of the stack before the reduce step. This guarantees that it becomes (the last) daughter of a rule if possible. The stack also guarantees that constituents (both lexical and nonlexical) occur in the order specified by the grammar. This guarantess not only that word order is correct but the fixed order means that a mother can be built out of a given set of daughters in only one way. Thus, the stack is very important for maintaining minimality. We will see that this is important in the next section where we generalise the shift-reduce parser to a parallel tabular variant.

6 A parallel tabular "shift-reduce" parser

In this section, we'll present a parallel tabular "shift-reduce" parser.[8] The parser is tabular since it uses a chart as a well-formed substring or lemma table. It is parallel or breadth-first since it pursues all "shift" and "reduce" steps in parallel. This will become clearer momentarily.

Before we describe the algorithm, we'll describe the chart "edges". An edge `edge(Sign,Beg,End)` contains a sign `Sign`, the string position `Beg` of the first word of the string that it spans and the string position `End` of the last word of the string that it spans. Therefore, if the first word in the input string is *the*, then the edge `edge(WSign,1,1)` will appear in the chart where `WSign` is the lexical entry for *the*. Similary, if the first two words of the input string are *the girl* and `NPSign` is the sign derived from parsing *the girl* then the edge `edge(NPSign,1,2)` will appear in the chart. That said, it is clear that a running count will need to be kept to indicate the string position of the next word in the input string.

`parse/2` takes two arguments, a list of words representing the input string and the current string index. `parse/1` just takes the entire input list and passes it to to `parse/2` with the string index `1`. The first clause is the termination case. If there are no more words left to process, the parser stops. We will see later that the chart will then contain every possible analysis of every substring of the input string.

```
parse(L) :-
  parse(L,1).

parse([],_).
parse([Word|Words],Num) :-
  (
  lexicon(Word,Sign),
  reduce(Sign,Num,Num),
```

[8] The first parser I have seen of this type was written by Jo Calder. His motivation was to write a parser for UCG grammars which did not involve a restriction-based prediction step since UCG rules contain almost no information which could be put to such use. A brief description of his parser is given in [4]. The version I will present here is a reconstruction of Calder's parser. Calder's parser is also rather ingenious since the design of the data structures allows the grammar rules to be compiled instead of interpreted.

```
fail ;
Num1 is Num + 1,
parse(Words, Num1)
).
```

The second clause has two very important functions which are crucial to making the whole algorithm work. First, the first word in the input list is looked up in the lexicon. The sign `Sign` is then passed with the current string index to `reduce`. So far this is perfectly analogous to the definition of `parse` for the standard shift-reduce parser. However, the next goal is `fail` which causes `reduce` to do every reduction possible. This includes "shifting" the lexical entry into the chart and successively trying to apply every rule. Thus, it pursues all shift and reduce steps before considering the next word in the input string. It is in this sense that the algorithm is parallel. When there are no more reduce steps left to try, the second disjunct is pursued. As in the standard algorithm, it recursively calls `parse` with the rest of the input string, and in this case, the string index of the first word on the rest of the input string. So, it is clear that the standard and parallel versions are essentially the same, except that the standard algorithm works depth-first and manipulates a stack while the tabular version works breadth-first and manipulates a chart.

The predicate `reduce/3` is also very similar to that of the standard algorithm (modulo data structures). The first argument `Kid` is the new constituent just built (as in the standard algorithm). The second and third arguments are the beginning and ending string indexes of the substring that the constituent spans.

As before, the first clause is the "shift" clause. However, instead of adding the new constituent to the stack, it adds it to the chart. The second clause is very much like that of the original algorithm. First, a rule is nondeterministically chosen. Then the daughters `Kids` of the rule are reversed and the "new" constituent `Kid` is required to unify with the first daughter of the reversed list (or to put it another way, to unify with the last daughter of the original list). This requirement is equivalent to pushing `Kid` on the stack in the standard algorithm before calling `find_kids`. Basically, we are insisting that if we can apply a rule, then `Kid` must be its last daughter. Again, this preserves minimality and completeness. Since we know that `Kid` is in the chart and unifies with the first daughter (in the reversed daughters list), we call `find_kids` with the rest of the reversed daughters `RevKids`. If all the daughters are successfully found, then we call `reduce` recursively with the mother sign of the rule `Mom`. So it is clear that the structure of `reduce` is essentially the same as that in the standard algorithm.

We haven't discussed the role of string indexing in `reduce` yet. `Beg2` and `End2` are the string positions of `Kid`. So, we want to start looking for `RevKids` at position `Beg2-1`. We call this `End1` for clarity since the last constituent will occur immediately before `Beg2`. Therefore, `find_kids` is called with `End1` as the third argument and the variable `Beg1`. It will become instantiated to the left string index of the left-most daughter. Thus, the mother sign will span `Beg1 - End2` and so those are the two indexes passed to the recursive call to `reduce` on the mother sign `Mom`.

```
reduce(Kid, Beg, End) :-                % shift
  assertz(edge(Kid, Beg, End)).
reduce(Kid, Beg2, End2) :-              % reduce
  rule(Mom, Kids),
  reverse(Kids, [Kid|RevKids]),
  End1 is Beg2 - 1,
  find_kids(RevKids, Beg1, End1),
```

```
reduce(Mom, Beg1, End2).
```

As the lowest level predicate responsible for manipulating the data structures (the chart in this case), `find_kids/3` is rather different than its counterpart in the standard algorithm. We discuss the second clause first. First, the first daughter `Kid` is looked for in the chart. `End2` specifies what the right end string index should be. This will return the left end string index of that daughter. So as in `reduce`, `End1` is set to `Beg2` − 1 for the recursive call to `find_kids` with the rest of the daughters `Kids`. Because `find_kids` is always called with the third argument decremented by 1 from the left of the last constituent, when all the daughters have been found, its value is 1 lower than what should be passed back. Therefore, the first clause increments the left index by 1 when there are no more daughters.[9]

```
find_kids([],Beg,Beg1) :-
  Beg is Beg1 + 1.
find_kids([Kid|Kids],Beg1,End2) :-
  edge(Kid,Beg2,End2),
  End1 is Beg2 - 1,
  find_kids(Kids,Beg1,End1).
```

We'll now present a generalised version of the parallel tabular "shift-reduce" parser. This time it is insuffient to use beg and end indexes to represent the substring spanned by a constituent since constituents might span a "discontinuous" substring of the input string. Therefore, we use *codes* to encode the lexical entries which have contributed to the derivation of a constituent. For an input string of length n, each code will be a bit pattern n bits long. If the ith bit of a constituent's code is 0 (for $1 \leq i \leq n$), it indicates that the constituent does not contain the ith word. For each word of index j that a constituent does contain, the jth bit will be 1.[10]

Let us say that two codes are ***bitwise-disjoint*** iff the logical AND of the two codes is 0. Furthermore, if A and B are two constituents, then they are bitwise-disjoint unless one is a *descendant* of the other. A is a descendant of B if $A = B$, A is a daughter of B or A is a descendant of a daughter of B. It therefore follows that the codes of all the daughters of a mother are bitwise-disjoint.

A lexical sign of string index i will be the bit string that is all 0s except for the ith bit which will be 1. The code of a mother sign will then be the logical OR of all the codes of its daughters. A string of length n must be analyzable as some sign and have a code of n 1 bits. For reasons of convenience, bit strings will be encoded as Prolog lists of `0` and `1`. We will see shortly that this leads to some efficiencies.

`parse/1` is similar to that of the previous parser. In this case, however, `parse/1` calls `parse/2` with an initial code rather than the first string index. The initial code is the code for the first lexical item. It is created by the predicate `initial_code`.

```
parse(L) :-
  initial_code(L,Code),
  parse(L,Code).
```

[9] It is obvious that this parser is closely related to the CKY parser. Whereas the CKY parser builds all constituents of length n before it builds any of length $n+1$, the breadth-first parser builds all constituents possible from the first n lexical entries before it considers the $n+1$ lexical entry. For a reference to the CKY parser, see [24].

[10] This encoding is also used by Johnson in [7]. However, he uses an alternative presentation based on sets of pairs of integers which is equivalent and more convenient for his purposes.

parse/2 is exactly analogous to the previous one except that it deals with codes instead of string indexes. Therefore, in the second clause, instead of incrementing a string index, the second disjunct increments the code. For example, if Code is [0,0,1,0,0] then Code1 will be [0,0,0,1,0]. The predicate next_code/2 takes care of the incrementing. Notice also that reduce takes only two arguments instead of three. This is because the code does the work of both the second and third arguments of the previous version.

```
parse([Word|Words],Code) :-
  ( lexicon(Word,Sign),
    reduce(Sign,Code),
    fail ;
    next_code(Code, Code1),
    parse(T, Code1) ).
```

The reduce/2 predicate is very similar to the reduce/3 predicate of the previous algorithm. Modulo the change from indexes to codes, the first clause is the same. Things are also basically the same in the second clause. A rule is chosen nondeterministically. Then the input sign is required to be one of the daughters of the rule by the call to delete/3 (instead of the last daughter as in the previous version). As before find_kids/3 tries to find edges in the chart that unify with the remaining daughters. It returns MomsCode which is the code of the mother sign (as built from the daughters' codes as indicated above). Finally, reduce is called recursively with the mother sign Mom and its code MomsCode.

```
reduce(Sign,Code) :-                % shift
  assertz(edge(Sign,Code)).
reduce(Kid, Code) :-                % reduce
  rule(Mom, Kids0),
  delete(Kid,Kids0,Kids),
  find_kids(Kids,Code,MomsCode),
  reduce(Mom,MomsCode).
```

In the second clause of reduce above, find_kids is called with both the daughter Kid and its code Code. Code is an argument since the mother's code MomsCode is constructed as the logical OR of all its daughters. find_kids accomplishes this as it finds acceptable edges to unify with the rest of the daughters in the rule chosen.

The first clause is just the standard base case which returns the completed mother's code when all the daughters have been found. In the second clause, we find an input list of daughters, a "cumulative" code Code0 which has been built so far and an output code Code. make_mask/2 takes as input the cumulative code and creates a "mask". This mask contains anonymous variables where Code0 contains 0s and 0s where Code0 contains 1s. For example, if Code0 is [0,1,1,0,1] then the mask KidsCode would be [_,0,0,_,0]. This means that the mask will not unify with any code which contains a 1 somewhere that Code0 does. This prevents lexical entries from occurring in two different edges A and B unless A is a descendant of B or vice versa. KidsCode is then used as the (partial) code for the first daughter in the list Kid when it is searched for in the chart. This will prevent it from unifying with any edges that it shouldn't but will also further instantiate the mask KidsCode completely. or_code then does a logical OR on Code0 and KidsCode producing Code1 which is then used as the cumulative code for the tail recursive call to find_kids. The use of the mask and the "cumulative" code gives an efficient way to use Prolog

unification to limit search in the chart. After all the daughters have been found, the mother's code is precisely the cumulative code and it is returned by the first clause.

```
find_kids([],Code,Code).
find_kids([Kid|Kids],Code0,Code) :-
  make_mask(Code0,KidsCode),
  edge(Kid,KidsCode),
  or_code(Code0,KidsCode,Code1),
  find_kids(Kids,Code1,Code).
```

7 A generalised CKY parser

In this section, we'll present a parser which implements the Cocke-Kasami-Younger (CKY) parsing algorithm. The CKY algorithm is a tabular algorithm which builds all phrases of length i before it builds any of length $i+1$ for $1 \leq i < n$ for a string of length n. For context-free grammars, edges are of the form $V(i,j)$ where V is a nonterminal symbol which derives the substring between string position i and j (inclusive). (Of course we will generalise this to arbitrary phrase structure grammars below.)

As in the parallel tabular shift-reduce parser, edges are of the form `edge(Sign,Beg,End)` where `Sign` is a sign and `Beg` and `End` are the string positions of the first and last words of the substring derived.

`parse/1` takes an input list and calls `initialise/3` with the input list and an initial word count of `0`. It returns the length of the list in `N`. `initialise/3` adds edges of the form `edge(Sign,I,I)` to the chart for each word in the input list with string position `I` between 0 and n. `parse/2` is then called with the minimum *span* `0` and the maximum span `N` to parse the lexical edges in the chart. The span is the length of the substrings to be parsed.

```
% parse(List)

parse(List) :-
  initialise(List,0,N),
  parse(1,N).

% initialise(List,First,Last)

initialise([],N,N).
initialise([Word|Words],N0,N) :-
  (
  lexicon(Word,Sign),
  assert(edge(Sign,N0,N0)),
  fail ;
  N1 is N0 + 1,
  initialise(Words,N1,N)
  ).
```

`initialise` is similar to the corresponding predicate in the parallel tabular shift-reduce parser. For the $i+1$th word `Word` in the input list, it asserts edges of the form `edge(Sign,N,N)` where

N = i for each lexical entry `lexicon(Word,Sign)`. It returns the length of the list in the last argument.

`parse/2` takes the current and maximum spans (Span0 and Span) and calls `parse_left_to_right/2` to parse all substrings of length Span0. It then increments the current span Span0 and calls itself recursively until the maximum span has been reached, i.e., the length of the input string.

```
% parse(Span0,Span)

parse(Span0,Span) :-
  Span0 > Span.
parse(Span0,Span) :-
  parse_left_to_right(Span0,Span),
  Span1 is Span0 + 1,
  parse(Span1,Span).
```

`parse_left_to_right/2` takes the current span Span0 and the maximum span Span and then passes the argument sequence `(0,Span,Span0)` to `parse_left_to_right/3` corresponding to the parameter sequence `(Beg,End,Span)` where Beg and End are the first and last indexes under consideration and Span is the current span.

`parse_left_to_right/3` starts at the left edge of the chart (string index 0) and tries to build phrases of length Span (the current span). It then iterates one string position at a time to the right checking for phrases of length Span until Beg + Span − End − 1 > 0, i.e., until no more phrases of length Span can be built. It calls `parse_substring/2` to parse every phrase between string positions Beg and Beg + Span at each step.

```
% parse_left_to_right(Span0,Span)

parse_left_to_right(Span0,Span) :-
  parse_left_to_right(0,Span,Span0).

% parse_left_to_right(Beg,End,Span)

parse_left_to_right(Beg,End,Span) :-
  Right is Beg + Span - 1,
  Right > End.
parse_left_to_right(Beg,End,Span) :-
  Right is Beg + Span - 1,
  parse_substring(Beg,Right),
  Beg1 is Beg + 1,
  parse_left_to_right(Beg1,End,Span).
```

`parse_substring/2` is very simple. Given string positions Beg and End, it nondeterministically choses a rule and then calls `find_kids/3` to find a phrase between Beg and End. It then asserts the mother sign Mom in the chart with those strong indexes. The `fail` guarantees that the chart will be complete between Beg and End.

```
% parse_substring(Beg,End)
```

```
parse_substring(Beg,End) :-
  rule(Mom,Kids),
  find_kids(Kids,Beg,End),
  assert(edge(Mom,Beg,End)),
  fail.
parse_substring(_,_).
```

Finally, `find_kids/3` takes a list of daughter signs `List` and beginning and ending indexes `Beg` and `End` and tries to find a contiguous set of edges which unify with the daugther signs. The first clause guarantees that the end of the last daughter in the phrase is `End`.

```
% find_kids(List,Beg,End)

find_kids([],Beg0,Beg) :-
  Beg is Beg0 - 1.
find_kids([Kid|Kids],Beg1,End2) :-
  edge(Kid,Beg1,End1),
  Beg2 is End1 + 1,
  find_kids(Kids,Beg2,End2).
```

This finishes the presentation of the parser. Before we consider minimality, soundness and completeness, we will first discuss the efficiency (or rather the lack of it) of this implementation of the CKY algorithm. The inefficiency of this implementation is due to the definition of `find_kids`. Assume that the current span is i. Then for every j s.t. $0 < j < i$, the entire search space for the span j will be researched. The only thing that prevents phrases previously built from being added to the chart again is the first clause of `find_kids` which requires that the ending string index of the phrase equals the ending index of the original call to `find_kids`.

Minimality follows from this property of `find_kids` and the definition of `parse_substring` which is minimal assuming `find_kids` is minimal. Soundness follows from the definition of `find_kids` which is similar to th e corresponding definition for the tabular parallel shift-reduce parser except that the latter works from "right to left" whereas the CKY algorithm works from "left to right". Completeness is very easy to see. We revert to our earlier notation temporarily. Assume that the chart is complete for all substrings between i and j. Then, to quote from [?, p187]

> "Since the table is complete for all substrings of the string between i and j, we merely need to check each rule, say, $A \longrightarrow B_1 \ldots B_n$ in G and look for $n+1$ positions k_0 through k_n such that $i = k + 0$ and $k_n = j$ and each k_i is greater than k_{i-1}, and such that the B_m are in the table under $V(k_{m-1}, k_m)$. If such a set of positions exists, then A can be added to the table under $V(i,j)$. By performing the search for rules and positions in all possible ways, we can complete the table for $V(i,j)$, in which case larger strings can be analyzed.
>
> Thus the CKY parsing algorithm builds the table by look for phrases of type $V(i,j)$ for larger and larger $j - i$.

We will now present a generalised version of the CKY algorithm which uses the *codes* of the generalised tabular parallel shift-reduce parsers. The overall structure is similar to both the former and the latter. Chart edges are of the form `edge(Sign,Code)` where `Sign` is a sign and `Code` is its code.

parse/1 takes an input list List and calls initial_code to create an initial Code as described for the generalised tabular shift-reduce parser. initialise adds every lexical entry for every word in the input list to the chart with its code returning the length of the list in Span. Then as with the previous algorithm it calls parse/2 with the minimal span 1 and the maximum span Span.

```
parse(List) :-
  initial_code(List,Code),
  initialise(List,0,Span,Code),
  parse(1,Span).
```

initial_code/2 is a little different from its previous version. Not only does it return an initial code of the form [1,0,...] but also asserts a clause of the form zero_mask(ZeroMask) where ZeroMask is a code of all zeros which is of length n for input string of length n. (zero_mask(N,ZeroMask) creates a code ZeroMask of all zeros of length N.)

```
% initial_code(List,Code)

initial_code(List,Code) :-
  length(List,N),
  zero_mask(N,ZeroMask),
  assert(zero_mask(ZeroMask)),
  M is N - 1,
  zero_mask(M,ZeroMask2),
  Code = [1|ZeroMask2].
```

initialise/4 is again slightly different than the previous version. In addition to counting the number of elements in the input list, it also takes an input code which is stored as part of the edge. The second disjunct in the second clause not only increases the word count but increments the code as described for the generalised tabular shift-reduce parser.

```
% initialise(List,First,Last,Code)

initialise([],N,N,Code).
initialise([Word|Words],N0,N,Code) :-
  (
  lexicon(Word,Sign),
  assert(edge(Sign,Code)),
  fail ;
  N1 is N0 + 1,
  next_code(Code,Code1),
  initialise(Words,N1,N,Code1)
  ).
```

parse/2 has roughly the same structure and function as its previous counterpart except that it calls parse_substring/1 with the current span Span rather than parse_left_to_right.

```
% parse(Span0,Span)
```

```
parse(Span0,Span) :-
  Span0 > Span.
parse(Span0,Span) :-
  parse_substring(Span0),
  Span1 is Span0 + 1,
  parse(Span1,Span).
```

parse_substring/1 has the same purpose as its previous counterpart but only needs the span information since daughters need not be contiguous. As before, rules are chosen nondeterministically and find_kids is called to find suitable daughter edges. The mother edge is then asserted.

```
% parse_substring(Span)

parse_substring(Span) :-
  rule(Mom,Kids),
  find_kids(Kids,Code,Span),
  assert(edge(Mom,Code)),
  fail.
parse_substring(_).
```

find_kids/3 takes a list of daughter signs List, returns the code of the mother sign (as explained in the description of the parallel tabular shift-reduce algorithm) in Code and also takes the current span as its third argument. It then creates an initial mask of all 0s and calls find_kids/4 to try to find the daughters.

find_kids/4 basically works like its correlate in the tabular parallel shift reduce parser. However, in addition to finding edges and manipulating codes, it incrementally checks that the number of words derived by the phrase being built does not exceed the current span. Furthermore, if we successfully find edges which unify with every rule daughters we check that the number of words that it "spans" is equal to the current span.

```
% find_kids(List,Code,Span)

find_kids(Kids,Code,Span) :-
  zero_mask(ZeroMask),
  find_kids(Kids,ZeroMask,Code,Span).

find_kids([],Code,Code,Span) :-
  number_of_ones(Code,Span).
find_kids([Kid|Kids],Code0,Code,Span) :-
  make_mask(Code0,KidsCode),
  edge(Kid,KidsCode),
  or_code(Code0,KidsCode,Code1),
  number_of_ones(Code1,Num),
  Num =< Span,
  find_kids(Kids,Code1,Code,Span).
```

This finishes the presentation of the generalised parser. It is even more inefficient than the definition of the standard parser due to the definition of find_kids and the fact that substrings

need not be continuous. I could have provided much more efficient implementations of the parsers but they would have been unsuitably obscure for pedagogical purposes.

Minimality and soundness are preserved for the same reasons that they are in the standard algorithm and completeness is also guaranteed since every substring of length i is analyzed before any strings of length $i+1$.

8 Computational complexity

Obviously, the computational complexity of each of the generalized algorithms will depend on the complexity of the algorithm being generalized. For the algorithms which nondeterministically delete an element of the input string, in the worst case, there are $n!$ possible permutations of the input string for an input string of length n. So, if we consider the top-down algorithm (which is exponential in the worst case, i.e., $\mathcal{O}(2^n)$), then the complexity will be $\mathcal{O}(n! \cdot 2^n)$. For the generalized tabular parser, we can construct grammars which will produce edges which span every subset of the input string, i.e., which parse the powerset of the input string. Thus the complexity is at least exponential in both time and space.

We could go on like this and try to do a complexity analysis for each generalised algorithm which shows how much worse it is than the standard algorithm. However, this seems like a waste of time since we are interested in the complexity of the "nonconcatenative" grammars themselves and not the parsing algorithms per se. Rounds [19] and Vijay-Shanker, Weir and Joshi [23] have done some interesting work in precisely this area. Rounds ([19]) defines two classes of "logic for parsing", ILFP and CLFP. He proves that a language is ILFP-defineable iff it is in *PTIME* (polynomial time parseable). ILFP is a formalism which characterises acceptability in terms of integers and arithmetical operations on them. He gives examples where the head-wrapping operations of [13] are characterised in terms of ILFP. On the other hand, CLFP characterises acceptability in terms of strings and concatenation. It defines precisely the class of languages which are in *EXPTIME* (exponential time parseable).

Vijay-Shanker et al. [23] examine classes of grammar formalisms which they characterise as ***linear context-free rewriting systems (LCFRS)***. The composition operators of LCFRSs are *linear* (do not duplicate unboundedly large structures) and ***nonerasing*** (do not erase unbounded structures). Furthermore, choices during a derivation are independent of the context of the derivation.

For the purpose of showing that LCFRSs are in *PTIME*, Vijay-Shanker et al. make the additional assumption that the contribution of a derived structure to the input string can be specified by a bounded sequence of substrings of the input.

> "Since each composition operation is linear and nonerasing, a bounded sequence of substrings associated with the resulting structure is obtained by combining the substrings in each of its arguments using only the concatenation operation, including each substring exactly once."
>
> ...
>
> "A derived structure will be mapped onto a sequence $x_1, \ldots, x_i$ of substrings (not necessarily continguous in the input), and the composition operations will be mapped onto functions that can be defined as follows.[11]

[11] In order to simplify the following discussion we assume that each composition operation is binary. It is easy to generalize to the case of n-ary operations.

$$f(\langle x_1, \ldots, x_{n_1}\rangle, \langle y_1, \ldots, y_{n_2}\rangle) = \langle z_1, \ldots, z_{n_3}\rangle$$

> where each z_i is the concatenation of strings from x_js and y_ks. The linear and nonerasing assumptions about the operations ...require that each x_j and y_k is used exactly once to define the strings $z_1, \ldots, z_{n_3}$."[12]

Thus for context free grammars containing the following two types of productions (where the N_i are nonterminals and T is a terminal)

1. $N_0 \longrightarrow N_1, \ldots, N_n$
2. $N_0 \longrightarrow T$

there will be an associated composition operator of the form

$$f_{N_0 \longrightarrow N_1,\ldots,N_n}(x_1, \ldots, x_n) = x_1 \ldots x_n$$

(for $n > 0$) for each type (1) production and a composition operator of the form

$$f_{N_0 \longrightarrow T}(x) = x$$

for each type (2) production. Since the two types of operators are defined only in terms of concatenation of substrings, they fall into the class described above and therefore are in *PTIME*.

Now consider a definition of the concatenation or ***append*** operator over strings.

$$\begin{aligned} \mathit{append}(\epsilon, x) &= x \\ \mathit{append}(xy_1, y_2) &= x \cdot \mathit{append}(y_1, y_2) \end{aligned}$$

Notice that the definition is ***recursive*** but is defined purely in terms of string concatenation. Notice furthermore, that the definition is ***functional***, thus fulfilling the extra assumptions that Vijay-Shanker et al. make to guarantee polynomial time parseability.

Now consider the definition of the following slightly more general ***shuffle*** operation.[13]

$$\begin{aligned} \mathit{shuffle}(\epsilon, \epsilon) &= \epsilon \\ \mathit{shuffle}(xy_1, y_2) &= x \cdot \mathit{shuffle}(y_1, y_2) \\ \mathit{shuffle}(y_1, xy_2) &= x \cdot \mathit{shuffle}(y_1, y_2) \end{aligned}$$

Notice that this definition is also recursive and is defined purely in terms of string concatenation. However, the operator is not a true operator since it is not functional. That is, for two given input arguments s_1, s_2, there may be several outputs. For example if $s_1 = ab$ and $s_2 = cd$ then

[12] Quote taken from [23]. In the original the xs are xs.

[13] Cf. [1].

$$\begin{aligned}
&\textit{shuffle}(s_1, s_2, abcd)\\
&\textit{shuffle}(s_1, s_2, acbd)\\
&\textit{shuffle}(s_1, s_2, acdb)\\
&\textit{shuffle}(s_1, s_2, cabd)\\
&\textit{shuffle}(s_1, s_2, cadb)\\
&\textit{shuffle}(s_1, s_2, cdab)
\end{aligned}$$

It is easy to see that we can define grammars which shuffle the terminal strings of some daughters and concatenate others. This characterizes exactly the class of grammars described in [17] and [18]. Does this composition "relation" define languages in *PTIME*? Unfortunately, the answer appears to be no.

The following theorem has been brought to my attention by Bill Rounds (p.c.).[14]

Theorem 1 (Ogden, Riddle and Rounds) *Let L_1 and L_2 be two deterministic context free languages. Let $L = \{s | s_1 \in L_1$ and $s_2 \in L_2$ and $\textit{shuffle}(s_1, s_2, s)\}$. Then the recognition problem for L is $\mathcal{NP}$-complete.*

The following corollary follows immediately.

Corollary 2 *Let L_1 and L_2 be two context free languages. Let $L = \{s | s_1 \in L_1$ and $s_2 \in L_2$ and $\textit{shuffle}(s_1, s_2, s)\}$. Then the recognition problem for L is $\mathcal{NP}$-complete.*

Now, let $G_1 = (N_1, T_1, P_1, S_1)$ and $G_2 = (N_2, T_2, P_2, S_2)$ be two context free grammars, where for $\alpha \in \{1, 2\}$, N_α is the set of nonterminal symbols, T_α is the set of terminal symbols, P_α is the set of productions and S_α is the start symbol of G_α respectively and $L_1 = L(G_1)$ and $L_2 = L(G_2)$. For each production $p \in P_\alpha$, define a composition operator f_p as described above for context free grammars. Then define $G = (N_1 \uplus N_2 \cup \{S\}, T_1 \uplus T_2, P_1 \uplus P_2 \cup \{S \longrightarrow S_1, S_2\}, S)$ (where $\uplus$ is disjoint union) for some $S \notin N_1 \cup N_2$. Finally, define

$$f_{(S \longrightarrow S_1, S_2)}(x, y) \quad = \quad \textit{shuffle}(x, y)$$

Then clearly $L(G) = \{s | s_1 \in L_1$ and $s_2 \in L_2$ and $\textit{shuffle}(s_1, s_2, s)\}$. Therefore, the recognition problem for L is $\mathcal{NP}$-complete.

There are two immediate responses to this result if it is correct. First, the parsing problem for most unification formalisms is $\mathcal{NP}$-complete so this isn't necessarily too worrying. A characterisation of average case performance would be very useful but eludes me at the moment. Second, it appears that the difference between the polynomial time parseability of languages defined using concatenation or *append* and those which also use the *shuffle* relation depends entirely on whether the composition operators are functional or just relational. The work of Vijay-Shanker et al. implies that this is the case but there is no explicit proof of this that I am aware of. Rounds' work is also very relevant here but again a definitive proof one way or another eludes me. For now, I will have to leave this issue in its current rather confused state.

[14] The source for the original is supposed to be a paper by Ogden, Riddle and Rounds in *Principles of Programming Languages 1978*. Unfortunately, I have not been able to locate it yet.

9 Concluding Remarks

We've presented three classes of algorithms: (1) a class of top-down, depth-first (simple top-down) and bottom-up with top-down filtering (left-corner and head-corner) parsers which can be generalised to be permutation-complete by nondeterministically choosing *any* lexical entry in the input string instead of deterministically choosing the first; (2) a class of bottom-up, depth-first (pushdown) stack parsers (shift-reduce) which can be generalised by replacing the stack with a multiset and nondeterministically choosing daughters from the multiset instead of a sequence from the top of the stack and (3) a class of parallel tabular algorithms (Calder's and the CKY algorithms) which can be generalised by using *codes* encoding derived substrings of the input string instead of string indexes to indicate the substrings that a constituent derives and by nondeterministically choosing (possibly) nonadjacent edges in the chart instead of deterministically choosing adjacent chart edges.

For each class of algorithms we have shown that they are minimal, sound and complete. For the first class, this is easy to see since it is evident that the algorithms recognise exactly the permutation-closure of the languages recognised by the original algorithms and do not produce the same derivation trees for (an indexed) input string more than once.

For the second class, soundness holds of both algorithms essentially because they are both depth-first algorithms and thus produce nonoverlapping sets of possible daughter constituents (where nonoverlapping means that the daughters are pairwise-disjoint with respect to the substrings of the input string they derive). Furthermore, the stack algorithm ensures correct linear order because of the stack discipline. They are both minimal because the last constituent constructed (either lexical or phrasal) is guaranteed to be a daughter in the next rule. This means that the set of derivations which the constituent can be a daughter in are only tried once. Thus the algorithms are minimal. Furthermore, this restriction contributes to completeness by guaranteeing that the algorithm attempts to use every constituent constructed as a daughter in every rule in the rule set. We can see by induction that the stack algorithm is complete by virtue of the stack and that the multiset algorithm is complete by virtue of the fact that subsets of the multiset are matched against daughters of the rule in every possible way. Thus it recognises the permutation-closure of the language recognised by the stack algorithm.

For the third class, proof of the existence of the three properties is similar to that of the second class. Soundness holds because the algorithms index edges with either string indexes or codes and rule application uses the indexes or codes to guarantee that two daughters in a rule are nonoverlapping. The adjacency restriction on string indexes ensures correct linear order in the index algorithms. Calder's algorithm and its generalization are both minimal for the same reason that the second class is. They are both minimal because the last constituent constructed is guaranteed to be a daughter in the next rule. Nothing further need be said. As in the second class, this also contributes to proof of completeness. For the CKY algorithm and its generalization the definitions of `find_kids` and `parse_substring` guarantee minimality. For Calder's algorithm, it is easy to give an inductive proof which shows that if the algorithm is complete for the first n lexical entries in the input string then it is complete for the first $n + 1$ entries. For the CKY algorithm, it is easy to give an inductive proof which shows that if the algorithm is complete for all substrings of length n, then it is complete for all substrings of length $n + 1$.

We can also give a similar inductive proof for the generalisation of Calder's algorithm which shows that if the algorithm is complete for the first n lexica l entries (i.e., every analysis of every discontinuous substring of the first n lexical entries has been derived) then it is complete for the first $n + 1$ lexical entries. Therefore the generalisation recognises the permutation-closure of the

languages recognised by Calder's algorithm. A similar proof is available for the generalised CKY algorithm.

Although the task that we set out to address might have seemed a bit murky at the start, the solution for the first class of algorithms is indeed trivial. (In fact, we could have applied the same solution to the shift-reduce algorithm.) For the second and third classes of algorithms, however, I think it is fair to say that we have uncovered something significant. That is, we have located precisely why some standard algorithms are minimal and complete, and by implication, now have a greater understanding of how to guarantee minimality and completeness for any new class of algorithms that we might consider. We could repeat the same exercise for other algorithms but doing so would probably be pointless unless such algorithms used entirely different techniques for guaranteeing minimality and completeness.

Despite assertions to the contrary in the introduction, the question of specialising the permutation-complete algorithms still looms large. We know from [23] that there is a very natural class of languages which are parseable in polynomial time. Furthermore, we know from [19] exactly what the class of polynomial parseable languages is. It is rather crude, to say the least, to use a permutation-complete algorithm with a grammar which generates a polynomial time language. Clearly, more work needs to be done on "folding in" some of the low-level string combination operations into the parsers. For example, if we consider a language which uses the *shuffle* operator defined above, then the use of one of the permutation-complete algorithms intact amounts to a generate and test strategy at best. For grammars, like Pollard's Head Grammars, such a generate and test strategy is even more indefensible. However, for now, I will have to leave this for future research.

Acknowledgements

I would like to thank Jo Calder, Gertjan van Noord, Evelyn van de Veen and Pete Whitelock for helpful comments and discussion. Some of this material has been presented to the Parsing Workshop at the Centre for Cognitive Science, University of Edinburgh. This work was partly supported by Esprit Project 3175 DYANA, *Dynamic Interpretation of Natural Language.*

References

[1] **Alfred V. Aho and Jeffrey D. Ullman [1972].** *The Theory of Parsing, Translation and Compiling.* Prentice-Hall: Englewood Cliffs, New Jersey.

[2] **Emmon Bach [1979].** Control in Montague Grammar. *Linguistic Inquiry*, **10**(4), pp515–531.

[3] **Emmon Bach [1983].** On the relationship between word-grammar and phrase-grammar. *Natural Language and Linguistic Theory*, **1**, pp65–89.

[4] **Jonathan Calder, Marc Moens and Henk Zeevat [1986].** *A* UCG *Interpreter.* ESPRIT project 393 ACORD Deliverable T2.6, December, 1986.

[5] **Jonathan Calder, Mike Reape and Henk Zeevat [1989].** An algorithm for generation in unification categorial grammar. In the *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics*, Manchester, April, 1989, pp233–240.

[6] **David Dowty [1990]**. Towards a minimalist theory of syntactic structure. In the *Proceedings of the Symposium on Discontinuous Constituency*, The Institute for Language Technology and Artificial Intelligence, Tilburg, The Netherlands, January 25-27, 1990, pp33–72.

[7] **Mark Johnson [1985]**. Parsing with discontinuous constituents. In the *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, Chicago, July 8–12, 1985, pp127–132.

[8] **Martin Kay [1989]**. Head driven parsing. In the *Proceedings of Workshop on Parsing Technologies*, Pittsburgh, 1989.

[9] **Gertjan van Noord [1989]**. BUG: A directed bottom-up generator for unification based formalisms. *Working Papers in Natural Language Processing, Katholieke Universiteit Leuven, Stichting Taaltechnologie Utrecht*, **4**, 1989.

[10] **Gertjan van Noord [1990]**. An overview of head-driven bottom-up generation. In Robert Dale, Chris Mellish and Michael Zock (eds.), *Current Research in Natural Language Generation.* Academic Press, 1990.

[11] **Gertjan van Noord [1991]**. Head Corner Parsing for Discontinuous Constituency. In the *Proceedings of the 29th Meeting of the Association for Computational Linguistics*, University of California, Berkeley, California, 18-21 June 1991, pp114–121.

[12] **Fernando C.N. Pereira and Stuart M. Shieber [1987]**. *Prolog and Natural-Language Analysis.* CSLI Lecture Notes Number 10, CSLI/Stanford, Stanford, California.

[13] **Carl Pollard [1984]**. *Generalized Context-Free Grammars, Head Grammars, and Natural Language.* PhD thesis, Stanford University, Stanford, 1984.

[14] **Carl Pollard and Ivan Sag [1987]**. *Information-based Syntax and Semantics*, **1**, CSLI Lecture Notes Number 13, CSLI/Stanford, Stanford, California.

[15] **Derek Proudian and Carl Pollard [1985]**. Parsing head-driven phrase structure grammar. In the *23th Annual Meeting of the Association for Computational Linguistics*, Chicago, July 8–12, 1985, pp167–171.

[16] **Mike Reape [1989]**. A logical treatment of semi-free word order and bounded discontinous constituency. In the *Proceedings of the 4th Meeting of the European Chapter of the ACL*, Manchester, England, April, 1989.

[17] **Mike Reape [1990]**. Getting Things in Order. In the *Proceedings of the Symposium on Discontinuous Constituency*, The Institute for Language Technology and Artificial Intelligence, Tilburg, The Netherlands, January 25-27, 1990, pp125-137.

[18] **Mike Reape [1990]**. A Theory of Word Order and Discontinuous Constituency in West Continental Germanic. In E. Engdahl and M. Reape (eds.), *Parametric Variation in Germanic and Romance: Preliminary Investigations.* ESPRIT Basic Research Action 3175 DYANA, Deliverable R1.1.A, January, 1990, pp25-40.

[19] **William C. Rounds [1988]**. LFP: A Logic for Linguistic Descriptions and an Analysis of its Complexity. *Computational Linguistics*, **14**(4), December, 1988.

[20] **Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira [1989].** A semantic-head-driven generation algorithm for unification based formalisms. In the *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, 1989.

[21] **Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira [1990].** Semantic-head-driven generation. *Computational Linguistics*, **16**(1), 1990.

[22] **Evelyn van de Veen [1990].** *Discontinuous Constituency and Parsing.* MSc thesis, Department of Artificial Intelligence, University of Edinburgh.

[23] **K. Vijay-Shanker, David J. Weir and Aravind K. Joshi [1987].** Characterizing structural descriptions produced by various grammatical formalisms. In the *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, Stanford, 1987.

[24] **David H. Younger [1967].** Recognition and parsing of context-free languages in time n^3. *Information and Control*, **10**, pp189–208.

A Appendix: an example

A.1 The grammar

First, we define a simple HPSG–style grammar. We first define the symbols VP and S as follows.

VP $=_{\text{def}}$ V[SUBCAT⟨NP[NOM]⟩]
S $=_{\text{def}}$ V[SUBCAT⟨⟩]

Next, consider the following lexical entries.

es: 'it'	NP[ACC]
ihm: 'him'	NP[DAT]
jemand: 'someone'	NP[NOM]
zu lesen: 'to read'	V[ZU, SUBCAT ⟨NP[NOM], NP[ACC]⟩]
versprochen: 'promised'	V[PSP, SUBCAT ⟨NP[NOM], NP[DAT],VP[ZU]⟩]
hat: 'has'	V[FIN, SUBCAT ⟨NP[NOM], VP[PSP]⟩]

Finally, consider the following two rule schemata.

$$1.\quad \text{S} \rightarrow \text{V[FIN,SUBCAT}\langle x_1,\ldots,x_n\rangle], x_1,\ldots,x_n$$
$$2.\quad \text{VP} \rightarrow \text{V[-FIN, SUBCAT}\langle \text{NP[NOM]}, x_2,\ldots,x_n\rangle], x_2,\ldots,x_n$$

We will now consider the following German subordinate clause.

(daß) es ihm jemand zu lesen versprochen hat
(that) it him someone to read promised has
'(that) someone promised him to read it'

A.2 The generalised shift-reduce parser

Here we give a derivation for the preceding example using the multiset parser. Each step is indicated by an 'S' for "shift" or by 'Rn' (for $n \in \{1,2\}$) for a "reduce" by rule n. Each step of the derivation is numbered and the multiset given. Each multiset contains elements which are pairs of *syntactic category:phonology*.

Step	Multiset	S/R?
1.	{NP[ACC]: es}	S
2.	{NP[ACC]: es, NP[DAT]: ihm}	S
3.	{NP[ACC]: es, NP[DAT]: ihm, NP[NOM]: jemand}	S
4.	{NP[ACC]: es, NP[DAT]: ihm, NP[NOM]: jemand, V[ZU]: zu lesen}	S
5.	{NP[DAT]: ihm, NP[NOM]: jemand, VP[ZU]: es zu lesen}	R2
6.	{NP[DAT]: ihm, NP[NOM]: jemand, VP[ZU]: es zu lesen, V[PSP]: versprochen}	S
7.	{NP[NOM]: jemand, VP[PSP]: es ihm zu lesen versprochen}	R2
8.	{NP[NOM]: jemand, VP[PSP]: es ihm zu lesen versprochen, V[FIN]: hat}	S
9.	{S[FIN]: es ihm jemand zu lesen versprochen hat }	R1

A.3 The parallel tabular "shift-reduce" parser

Next, we present a derivation for the same example using the parallel tabular "shift-reduce" parser. Each step corresponds to adding another edge to the chart. Thus, the edges are listed in order of occurrence. The syntactic category, phonology and code for each edge is listed. Where an edge was built by a reduction step, the rule number is given.

Edge	Category	Phonology	Code	Rule
1.	NP[ACC]	es	[1,0,0,0,0,0]	
2.	NP[DAT]	ihm	[0,1,0,0,0,0]	
3.	NP[NOM]	jemand	[0,0,1,0,0,0]	
4.	V[ZU]	zu lesen	[0,0,0,1,0,0]	
5.	VP[ZU]	es zu lesen	[1,0,0,1,0,0]	2
6.	V[PSP]	versprochen	[0,0,0,0,1,0]	
7.	VP[PSP]	es ihm zu lesen versprochen	[1,1,0,1,1,0]	2
8.	V[FIN]	hat	[0,0,0,0,0,1]	
9.	S[FIN]	es ihm jemand zu lesen versprochen hat	[1,1,1,1,1,1]	1