Remarks on the Bottom-up Generation Algorithm

Herbert Ruessink
Gertjan van Noord
OTS RUU Utrecht
ruessink@hutruu59.BITnet
vannoord@hutruu59.BITnet

1990

Abstract

In this squib we wish to draw attention to two issues related to the generation algorithm described in Shieber et al. (1989) and Shieber et al. (1990). The first issue concerns a simple transformation by which a DCG grammar can be *compiled* into a generator which is equivalent to the interpreter described in Shieber et al. (1989) and Shieber et al. (1990). The second issue concerns the common architecture of the bottom-up generation algorithm and the bottom-up parsing algorithm described by Matsumoto et al. (1983).

1 A Compiled DCG Generator

The bottom-up generation procedure described in [3, 4] is presented solely as an interpreter for a DCG grammar. However, it is possible to perform a simple transformation of the rules to derive a compiled version of the grammar, similar to the transformation performed by the standard DCG compiler included in many Prolog implementations (cf. also the compiled bottom-up parser described by Matsumoto et al in [1]). Compiled grammars are in general preferred because of their increased efficiency.

Note that the bottom-up procedure distinguishes between two types of procedure which each use a separate set of rules. $generate/1$ uses the rules selected by $applicable\_non\_chain\_rule/2$ and $connect/2$ uses the rules selected by $applicable\_chain\_rule/4$. The compilation procedure will therefore be expected to produce two types of predicate, one to replace $generate/1$ and another to replace $connect/2$.

The compilation procedure will translate rules of the grammar into Prolog predicates. Let us first consider the translation of a chain rule into the compiled equivalent of *connect*/2.

A chain rule was defined as a rule "in which the semantics of some right-hand-side element is identical to the semantics of the left-hand side" [4]. This can be schematically represented as follows:

```
M/Sh ---> Da/Sa .... Dh/Sh .... Dn/Sn
```

This rule is translated into the following clause for the predicate *c*/2:

```
c( Dh/Sh, Top ) :-
    chained_nodes( M/Sh, Top ),
    g( Da/Sa ),
      .
      .
    g( Dg/Sg ),
    g( Di/Si ),
      .
      .
    c( M/Sh, Top ).
```

A non-chain rule is transformed into a clause for the predicate g/1. Note that in a non-chain rule the semantics of the left-hand side are not identical to those of any element on the right-hand side:

```
M/S ---> Da/Sa .... Dn/Sn
```

The schematic form of the corresponding clause is:

```
g( Top/S ) :-
   chained_nodes( M/S, Top/S ),
   g( Da/Sa ),
     .
     .
   g( Dn/Sn ),
   c( M/S, Top/S ).
```

We shall show the relation of these compiled predicates to the interpreted procedure by instantiating the predicates *generate*/1 and *connect*/2 for a specific rule (the following should not taken as describing the actual manner in which the compiled predicates are derived). Consider for instance the non-chain rule

```
vp(finite, [np(3-sing)/S])/leave(S) ---> [leaves].
```

The automatic process mentioned in section 3.1 of Shieber et al (1990) converts this rule to a prolog clause using the difference list notation to represent the string positions:

```
non_chain_rule(
   node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
   []          ).
```

Partial execution of the predicates *applicable_non_chain_rule*/3 and *generate*/1 leads to the following two Prolog clauses:

```
applicable_non_chain_rule(
  node( Top/leave(S), Y-X),
  node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
               []      ) :-
     node_semantics(
        node( Top/leave(S), Y-X),
        leave(S)   ),
     node_semantics(
        node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
        leave(S)   ),
     non_chain_rule(
        node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
        []         ),
     unify(
        node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
        node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P )
          ),
     chained_nodes(
        node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
        node( Top/leave(S), Y-X)
                        ).

generate( node( Top/leave(S), Y-X) ) :-
  applicable_non_chain_rule(
        node( Top/leave(S), Y-X),
        node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
        []                  ),
  generate_rhs( [] ),
  connect(
        node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
        node( Top/leave(S), Y-X)
         ).
```

The unresolved call(s) from *applicable_non_chain_rule*/3 can replace the call to that predicate in *generate*/1. Similarly the call to *generate_rhs*/1 can be replaced by the call(s) resulting from that predicate (in this case none):

```
g( node( Top/leave(S), Y-X) ) :-
    chained_nodes(
          node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
```

```
        node( Top/leave(S), Y-X)
               ),
  c( node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
     node( Top/leave(S), Y-X)
   ).
```

This clause is the compiled version of the non-chain rule.

The compilation of chain rules can be illustrated in the same manner. Consider the following chain rule and the Prolog clause derived from it:

```
s(Form)/S ---> Subj, vp( Form, [Subj])/S.

chain_rule(
            % the mother of the rule
            node( s( Form )/S, P0-PN ),
            % the non-head daughters
            [ node( Subj, P0-P1 ) ],
            % the semantic head of the rule
            node( vp( Form, [ Subj ] )/S, P1-PN )
          ).
```

Instantiating *applicable_chain_rule*/4 and *connect*/2 with this rule results in the following clauses:

```
applicable_chain_rule(
    node( vp( Form, [ Subj ] )/S, P1-PN ),
    node( s( Form )/S, P0-PN ),
    node( Top/S, X-Y ),
    [ node( Subj, P0-P1 ) ]
                    ) :-
     chain_rule( node( s( Form )/S, P0-PN ),
                 [ node( Subj, P0-P1 ) ],
                 node( vp( Form, [ Subj ] )/S, P1-PN )
               ),
     unify( node(vp( Form, [ Subj ])/S, P1-PN ),
            node(vp( Form, [Subj])/S, P1-PN)
          ),
     chained_nodes( node( s( Form )/S, P0-PN ),
                    node( Top/S, X-Y )
                  ).

 connect( node( vp( Form, [Subj])/S, P1-PN ),
          node( Top/S, X-Y ) ) :-
     applicable_chain_rule(
              node( vp( Form, [ Subj ] )/S, P1-PN ),
              node( s( Form )/S, P0-PN ),
              node( Top/S, X-Y ),
              [ node( Subj, P0-P1 ) ]
```

```
                             ),
          generate_rhs( [ node( Subj , P0-P1 ) ] ),
          connect( node( s( Form )/S, P0-PN ) ),
                   node( Top/S, X-Y )
                 ).
```

Again the calls to *applicable_chain_rule*/4 and *generate_rhs*/1 in the clause for *connect*/2 can be replaced by the unresolved calls of these clauses:

```
c( node(vp( Form, [ Subj ])/S, P1-PN) ),
   node( Cat/S, X-Y )
 ) :-
    chained_nodes( node( s( Form )/S, P0-PN ),
                   node( Cat/S, X-Y )
                 ),
    g( node( Subj, P0-P1 ) ),
    c( node( s( Form )/S, P0-PN ) ),
       node( Cat/S, X-Y )
          ).
```

Lastly it must be noted that the clause:

```
connect( Pivot, Root ) :-
         unify( Pivot, Root).
```

is essential to the procedure but has no corresponding rule in the grammar. It must therefore be added to the set of compiled rules to produce a complete system which is equivalent to the interpreted version.

Although the illustrations above do not correctly describe the actual compilation process of the grammar, it serves to clarify the relation between the interpreted and the compiled versions. An automatic procedure which performs the transformation of *chain_rule*/3 and *non_chain_rule*/2 clauses into a compiled generator has been implemented for the RUU Standard DCG Interpreter/Compiler and for the MiMo 2 Translation System.

2 A Common Architecture for Parsing and Generation

The architecture of the generation process described in [3, 4] and that of the bottom-up parsing process first described in [1] show striking similarities. For this discussion we will use the interpreter version [2] of the bup parser. In this section we will attempt to show that in fact the bottom-up generation and parsing procedures are in fact two instances of a single proof procedure for Horn clauses.

In this discussion, as in the first section, we will assume that the DCG rules are transformed into Prolog facts. A DCG rule such as:

```
vp(finite, [np(3-sing)/S])/leave(S) ---> [leaves].
```

will be recognized as a dictionary entry and will be transformed into the following Prolog fact:

```
word(
     node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P )
    ).
```

A syntactic rule which does not introduce a terminal, eg.:

```
s(Form)/S ---> Subj, vp( Form, [Subj])/S.
```

will be represented by:

```
rule(
       % the mother of the rule
       node( s( Form )/S, P0-PN ),
       % the daughters of the rule
       [ node( Subj, P0-P1 ),
         node( vp( Form, [ Subj ] )/S, P1-PN ) ]
     ).
```

The procedure by which the rules are recognized as being of either type will be discussed below. At this point we will just accept the distinction.

The parsing algorithm proposed by Matsumoto et al (1983) is illustrated by bup1:

```
bup1( LF, String ) :-
    goal( node( s/LF, String-[] ) ).

goal( Node ) :-
    predict_word( Small, Node ),
    up( Small, Node ).

up( Node, Node ).
up( Small, Big ) :-
    predict_rule( Small, Middle, Others, Big ),
    goal_ds( Others ), connect( Middle, Big ).

goal_ds( [] ).
goal_ds( [ Node | Nodes ] ) :-
     goal( Node ),
     goal_ds( Nodes ).

predict_word( node( W/LF, P0-P1 ), node( _/_, P0-_ ) ) :-
    word( node( W/LF, P0-P1 ) ).
```

```
predict_rule( Head, Mother, Others, _ ) :-
    rule( Mother, [ Head| Others ] ).
```

The first observation that we can make is that Matsumoto et al. make a distinction between two types of rules, *word*/1, which introduces terminals, and *rule*/2 which does not introduce a terminal. This is similar to the generation algorithm which distinguishes between non-chain rules and chain rules. On closer inspection we also find that the organization of the predicates resembles the generation algorithm, with one small difference. The predicate *generate*/1 expects a non chain rule to possibly have daughters. The parsing algorithm does not expect a node which expands into a terminal to have any daughters. However, it is not at all impossible that such rules arise in a DCG grammar:

```
category/sem ---> [word], category2/sem2.
```

We could extend the algorithm to deal with such rules by extending the predicate *word*/1 to *word*/2 where the second argument represents a list of daughters. For 'normal' lexical entries this argument will be the nil list '[]':

```
word( node( category/sem, [ word | P1 ]-PN ),
    [ node( category2/sem2, P1-PN ) ).

word(
   node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
   []
    ).
```

Extending the predicate *word*/1 into *word*/2 will require the following changes in the predicates *predict_word*/2 (which becomes *predict_word*/3) and *goal*/1:

```
predict_word( node( W/LF, P0-P1 ),
              node( _/_, P0-_ ),
              Others ) :-
    word( node( W/LF, P0-P1 ), Others ).

goal( Node ) :-
    predict_word( Small, Node, Others ),
    goal_ds( Others ),
    connect( Small, Node ).
```

The organization of the parsing procedure and the generation procedure is now identical. *goal*/1 corresponds to *generate*/1 and *up*/2 corresponds to *connect*/2. The remaining difference between the two procedures now resides in the rules, or to be more precise, in the manner in which the rules are recognized to belong to either of the two sets of rules that are used.

In the case of generation, the distinction between chain rules and non-chain rules is the fact that in chain-rules the semantics of a right-hand-side element is identical to the semantics of the left-had side, whereas this is not the case for non-chain rules. The semantics can be viewed as the 'leading' feature in recognizing a chain-rule. Note also that the input for generation is a specific instantiation of the semantics of a node. Assuming that this is not coincidental, we can generalize this fact by stating that the input is the 'leading' feature for a procedure, and that a chain-rule is identified by a daughter node which has the same leading feature as the mother of the rule.

In parsing the input is a string of words. Following the generalization we expect the string to be the leading feature for parsing. Recall that the nodes in the prolog facts incorporate a difference list notation to represent the part of the string that is covered by that node:

```
node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P )
node( s( Form )/S, P0-PN )
```

The input string is represented by the first element of the difference list. We therefore expect a chain rule to have a daughter node which has the same input as the mother node. Consider the following rule:

```
rule(
       node( s( Form )/S, P0-PN ),
       [ node( Subj, P0-P1 ),
         node( vp( Form, [ Subj ] )/S, P1-PN ) ]
    ).
```

Note that the input of the Subject node, P0, is identical to the input of the s. This is not the case for lexical entries:

```
word( node( category/sem, [ word | P1 ]-PN ),
      [ node( category2/sem2, P1-PN ) ).
```

In this case the input for the mother of the rule is $[word|P1]$, whereas for the daughter it is the string P1. The daughter obviously does not cover the initial part of the string covered by the mother. It will be clear that this is also not the case if no daughter is present at all, for instance in:

```
word(
   node( vp(finite, [np(3-sing)/S])/leave(S), [leaves|P]-P ),
   []
    ).
```

As in the case of generation, the leading feature for parsing can be used to distinguish chain-rules, represented by the predicate $rule/3$, from non-chain rules, represented by the predicate $word/3$.

The last, somewhat trivial observation we wish to make is that the selection of applicable non-chain rules in both procedures takes place on the basis of the leading feature. The predicate *applicable_non_chain_rule*/3 explicitly matches the semantics of the top node with those of the mother of the non-chain rule. Similarly, the predicate *predict_word*/3 selects the rules on the basis of the initial part of the string covered by the top node. This is not surprising, in both cases the input is represented as the leading feature of the top node.

The observation that the bottom-up parsing and generation procedures are identical when viewed in terms of leading feature leads to the conclusion that they are two instances of a single proof procedure for Horn clauses. A set of Horn clauses representing a DCG grammar can be regarded as defining a set of acceptable strings as well as as a set of acceptable logical representations (where the proof procedure for a string produces one or more logical representations as a side effect and vice versa). The proof procedure in both cases is based on the same algorithm. In order to prove the input, the procedure selects a non-chain rule whose mother node agrees with the input (the logical representation for generation and the string for parsing). It then proceeds in a top-down fashion to recursively prove the daughters of the selected rule. As a result the mother node of the rule will have been proven. Subsequently, the algorithm selects a chain-rule. This rule is selected on the basis of the head of the rule, the daughter which agrees with the mother of the chain rule with respect to the leading feature. This head is required to match the node that has already been proven at that point (the mother of the non-chain rule in this case). Having selected such a rule the procedure must only recursively prove the other daughters of the chain-rule in order to prove the mother. Having proved the mother, the algorithm can continue by selecting another chain rule using the same principles. The algorithm succeeds in proving the input when a node which matches the input has been derived.

References

[1] Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. Bup: a bottom up parser embedded in prolog. *New Generation Computing*, 1(2), 1983.

[2] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural Language Analysis*. Center for the Study of Language and Information Stanford, 1987.

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

[4] Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira. Semantic-head-driven generation. *Computational Linguistics*, 1990. To appear.