Head-Driven Generation and Indexing in ALE

Gerald Penn
SFB 340
Kl. Wilhelmstr. 113
72074 Tübingen, Germany
gpenn@sfs.nphil.uni-tuebingen.de

Octav Popescu
Computational Linguistics Program
Carnegie Mellon University
Pittsburgh, PA 15213, USA
octav@cs.cmu.edu

Abstract

We present a method for compiling grammars into efficient code for head-driven generation in ALE. Like other compilation techniques already used in ALE, this method integrates ALE's compiled code for logical operations with control-specific information from (SNMP90)'s algorithm along with user-defined directives to identify semantics-related substructures. This combination provides far better performance than typical bi-directional feature-based parser/generators, while requiring a minimum of adjustment to the grammar signature itself, and a minimum of extra compilation.

1 Motivation

Large-scale development systems for typed feature-based grammars have typically oriented themselves towards parsing, either ignoring generation entirely (the usual case), or assuming that generation can be achieved for free by using a bi-directional control strategy with a semantically, rather than phonologically, instantiated query. In the latter case, the result has inevitably been a system which is unacceptably slow in both directions. At the same time, several lower-level logical operations over typed feature structures, such as inferring a type from the existence of an appropriate feature, or the unification of two feature structures, are indeed common to both parsing and generation; and generators outside this logical domain, of course, can make no use of them. What is required is a system which provides a common pool of these operations optimized for this particular logic, while also providing modes of processing which are suited to the task at hand, namely parsing or generation.

This is exactly how the situation has developed in other areas of logic programming. The Warren Abstract Machine and its various enhancements are now the *de facto* standard for Prolog compilation, for example; and with that standard come techniques for call stack management, heap data structures etc.; but this does not mean that all Prolog programs are created equal — the more sophisticated compilers use *mode declarations* in order to optimize particular programs to being called with certain argument instantiations.

The Attribute Logic Engine (ALE,(CP94)) is a logic programming language based on typed feature structures, which can compile common logical operations like type inferencing and unification into efficient lower-level code. ALE also *compiles* grammars themselves into lower-level instructions, rather than simply running an interpreter over them, which yields a substantial increase in efficiency. For ALE, the question of efficient generation is then how to compile grammars for use in semantically-instantiated queries. To date, however, ALE has fallen within the class of systems which have ignored generation entirely. Its only control strategies have been a built-in bottom-up chart parser, and the usual SLD-resolution strategy for its Prolog-like language.

On the other hand, only a few of the operations it compiles are specific to the parsing direction. ALE's lower level of instructions are expressed using Prolog itself as the intermediate code. ALE compiles the various elements of a typed feature-based grammar (type signature, feature declarations, lexical rules, phrase structure-like grammar rules) into Prolog clauses which are then compiled further by a Prolog compiler for use at run-time. In fact, ALE also has a Prolog-like logic programming language of its own, based on typed feature structures. Goals from this language can be used as procedural attachments on lexical rules or grammar rules as well.

This paper describes a head-driven generator which has recently been added to ALE ((Pop96)), which provides a smooth integration of generation-specific control information with the powerful logical compilation that ALE already performs. We also briefly consider the use of a lexical indexing strategy for generation, which is compiled into efficient lower-level instructions as well.

2 Head-Driven Generation

Our head-driven generator uses essentially the same control strategy as proposed in (SNMP90), which was first used in the BUG system of (Noo89). This algorithm is quite well suited to large-scale HPSG generation, as it avoids the termination problems inherent to top-down processing of strongly lexicocentric theories, and, at the same time, does not require of its grammar rules the same naïve form of compositionality, known as *semantic monotonicity*, as Earley-based strategies do. A semantically monotonic grammar rule is one in which the semantic contribution of every daughter category subsumes a portion of the contribution of the mother category. In general, wide-coverage theories cannot guarantee this.

Control in this algorithm is guided by meaning rather than a particular direction over a string, and thus requires the user to distinguish two classes of rules: those in which a mother has the same semantics as some daughter (a *chain rule*), and those in which it does not (*non-chain rule*). The strategy is a combination of bottom-up and top-down steps based on the location of a *pivot*, the lowest node in a derivation tree which has the same semantics as the root goal. Once a pivot is located, one can recursively process top-down from there with non-chain rules (since the pivot must be the lowest such node), and attach the pivot to the root bottom-up with chain rules. A pivot can either be a lexical entry or empty category (the base cases), or the mother category of a non-chain rule. The base case for bottom-up processing is when the pivot and root are taken to be the same node, and thus unified. The reader is referred to (SNMP90) for the complete algorithm.

What we will be concerned with here is the adaptation of this algorithm to grammars based on a logic of typed feature structures, such as HPSG. (SNMP90) uses definite clause grammars, while (Noo89) uses a Prolog-based extension of PATR-II, which has features and atoms, but no feature-bearing types, and thus no appropriateness. Unlike both of these approaches, our goal is also to compile the grammar itself into lower-level code which is specifically suited to the particular requirements of head-driven generation, very much as ALE already does for its parser, and much as one would compile a Prolog program for a particular set of mode specifications.

3 Input Specification

The reader is referred to (CP94) for a complete specification of ALE's syntax as it pertains to parsing. ALE allows the user to refer to feature structures by means of *descriptions*, taken from a language which allows reference to types (Prolog atoms), feature values (colon-separated paths), conjunction and disjunction (as in Prolog), and structure sharing through the use of variables (with Prolog variables). ALE grammar rules simply consist of a series of these descriptions, one for each daughter and one for the mother, interspersed with procedural attachments from ALE's Prolog-like language. The following is a typical $S \rightarrow NP\ VP$ rule taken from a simple ALE grammar:

```
srule rule
 (s,phon:SPhon,form:Form,sem:S) ===>
  cat> (phon:SubjPhon),Subj,
  sem_head> (vp,phon:VpPhon,form:Form,
             subcat:[Subj],sem:S),
  goal> append(SubjPhon,VpPhon,SPhon).
```

The description of a sentence-typed feature structure before the **===>** is the description of the mother category. The operator, **cat>**, identifies a daughter description, here used for the subject NP, and **goal>** identifies a call to a procedural attachment, whose arguments are Prolog variables instantiated to their respective phonologies (the values of feature, **phon**). **sem_head>** is a new operator which identifies the daughter description corresponding to the semantic head of a rule, according to (SNMP90)'s definition. Grammar rules can have at most one **sem_head>** declaration; and those which have one are identified as chain rules.

The only other special information the user needs to provide is what constitutes the semantic component of a feature structure. ALE uses a distinguished predicate, **sem_select(+,-)**, from its procedural attachment language in order to identify this material, e.g.:

```
sem_select(sem:S,S) if true.
```

In general, this material may be distributed over various substructures of a given feature structure, in which case the predicate may be more complex:

```
sem_select((sign,synsem:cont:Cont,
                 retrieved_quants:QR),
           (sem,c:Cont,q:QR)) if
  no_free_vars(QR).
```

Notice that such grammars can still be compiled by ALE's parsing compiler: the **sem_select/2** predicate can simply be ignored, and a **sem_head>** operator can be interpreted exactly as **cat>**. In the general case, however, a particular grammar rule will not compile into efficient, or even terminating, code in both modes, particularly when procedural attachments are used. Just as in the case of Prolog, the user is responsible for ordering the procedural attachments (subgoals) with respect to their daughter categories and with respect to each other to ensure proper termination for a particular mode of processing. Just as in Prolog, one could also modify ALE to assist, to an extent, by augmenting ALE's procedural attachments with mode declarations which can be enforced by static analysis during compilation. At this point, one could also adapt techniques for automatic mode reversal from logic programming ((Str90; MGH93)) to grammar rules to obtain the minimum amount of manual modification necessary.

4 Compilation

All ALE compilation up to, and including, the level of descriptions applies to generation without change. This includes compiled type inferencing, feature value access functions, and the feature structure unification code itself.[1] This level is a very important and convenient stage in compilation, because descriptions serve as the basic building blocks of all higher-level components in ALE. One of these components, ALE's procedural attachment language, can also be compiled as in the parsing case, since it uses the same SLD resolution strategy. The rest are described in the remainder of this section.

4.1 Grammar Rules

Chain rules and non-chain rules are compiled differently because (SNMP90)'s algorithm uses a different control strategy with each of them. Both of them are different from the strategy which ALE's bottom-up parser uses. All three, however, vary only slightly in their use of building blocks of code for enforcing descriptions on feature structures. These building blocks of code will be indicated by square brackets, e.g. `[add Desc to FS]`.

4.1.1 Non-chain Rules:

Non-chain rules have no semantic head, and are simply processed top-down, using the mother as a pivot. We also process the daughters from left to right. So the non-chain rule:

[1](CP96) provides complete details about this level of compilation.

```
D0 ===> D1, ..., DN.
```

consisting of descriptions D0 through DN, is compiled to:

```
non_chain_rule(+PivotFS,+RootFS,?Ws,
               ?WsRest) :-
  [add D0 to PivotFS],
  exists_chain(PivotFS,RootFS),
  [add D1 to FS1],
  generate(FS1,SubWs,SubWs2),
  [add D2 to FS2],
  generate(FS2,SubWs2,SubWs3),
  ...
  [add DN to FSN],
  generate(FSN,SubWsN,SubWsRest),
  connect(PivotFS,RootFS,SubWs,SubWsRest,
          Ws,WsRest).
```

non_chain_rule/4 is called whenever a non-chain rule's mother is selected as the pivot (by successfully adding the mother's description, D0, to **PivotFS**), generating a string represented by the difference list, **Ws-WsRest**. The algorithm says one must recursively generate each daughter (**generate/3**), and then connect this pivot-rooted derivation tree to the root (**connect/6**). Before we spend the effort on recursive calls, we also want to know whether this pivot can in fact be connected to the root; this is accomplished by **exists_chain/2**. In general, the mother category and daughter categories may share substructures, through the co-instantiation of Prolog variables in their descriptions. After matching the mother's description, which will bind those variables, we add each daughters' description to a new structure **FSi**, initially a structure of type **bot** (the most general type in ALE), before making the respective recursive call. In this way, the appropriate information shared between descriptions in the user's grammar rule is passed between feature structures at run-time.

To generate, we use the user's distinguished selection predicate to build a candidate pivot, and then try to match it to the mother of a non-chain rule (the base cases will be considered below):

```
generate(+GoalFS,?Ws,?WsRest) :-
  solve(sem_select(GoalFS,Sem)),
  solve(sem_select(PivotFS,Sem)),
  non_chain_rule(PivotFS,GoalFS,Ws,WsRest).
```

solve/1 is ALE's instruction for making calls to its procedural attachment language. Its clauses are compiled from the user's predicates, which have description arguments, into predicates with feature structure arguments as represented internally in ALE.

4.1.2 Chain Rules:

Chain rules are used to connect pivots to goals. As a result, we use them bottom-up from semantic head to mother, and then recursively generate the non-head daughters top-down, left to right. So a chain rule:

```
D0 ===> D1, ..., DK, HD, D(K+1), ..., DN.
```

is compiled to:

```
chain_rule(+PivotFS,+RootFS,+SubWs,
           -SubWsRest,?Ws,?WsRest) :-
  [add HD to PivotFS],
  [add D0 to MotherFS]
  exists_chain(MotherFS,RootFS),
  [add D1 to FS1],
  generate(FS1,SubWs,SubWs2),
  ...
  [add DK to FSK],
  generate(FSK,SubWsK,SubWsK+1),
  [add D(K+1) to FS(K+1)],
  generate(FS(K+1),SubWsK+1,SubWsK+2),
  ...
  [add DN to FSN],
  generate(FSN,SubWsN,SubWsRest),
  connect(MotherFS,RootFS,SubWs,SubWsRest,
          Ws,WsRest).
```

chain_rule/6 is called whenever a chain rule is selected to connect a pivot (**PivotFS**) to a root goal (**RootFS**), yielding the string **Ws-WsRest**, which contains the substring, **SubWs-SubWsRest**. In the case of both chain and non-chain rules, calls to a procedural attachment between daughter **Di** and **D(i+1)** are simply added between the code for **Di** and **D(i+1)**. Procedures which attach to the semantic head, in the case of chain rules, must be distinguished as such, so that they can be called earlier.

To connect a pivot to the root, we either unify them (the base case):

```
connect(PivotFS,RootFS,Ws,WsRest,Ws,
        WsRest) :-
  unify(PivotFS,RootFS).
```

or use a chain rule:

```
connect(+PivotFS,+RootFS,+SubWs,-SubWsRest,
        ?Ws,?WsRest) :-
  chain_rule(PivotFS,RootFS,SubWs,
             SubWsRest,Ws,WsRest).
```

Similarly, to discover whether a chain exists, we either unify, or attempt to use one or more chain rules. For each chain rule, we can, thus, compile a separate clause for **exists_chain/2**, for which that rule is the last step in the chain. In practice, a set of chain rules may have potentially unbounded length chains. For this reason, we bound the length with a constant declared by the user directive, **max_chain_length/1**.

4.2 Lexical Entries

Lexical entries are the base cases of the algorithm's top-down processing, and can be chosen as pivots instead of the mothers of non-chain rules. In fact, lexical entries can be compiled exactly as a non-chain rule with no daughters would be. So a lexical entry for **W**, with description, **D**, can be compiled into the **non_chain_rule/4** clause:

```
non_chain_rule(PivotFS,RootFS,Ws,WsRest) :-
  [add D to PivotFS],
  connect(PivotFS,RootFS,[W|SubWs],SubWs,
          Ws,WsRest).
```

For ALE's bottom-up parser, lexical entries were compiled into actual feature structures. Now they are being compiled into code which executes on an already existing feature structure, namely the most general satisfier of what is already known about the current pivot. Empty categories are compiled in the same way, only with no phonological contribution. This method of compilation is re-evaluated in Section 6.

4.3 Lexical Rules

ALE's lexical rules consist simply of an input and output description, combined with a morphological translation and possibly some procedural attachments. In this present third singular lexical rule:

```
pres_sg3 lex_rule (vp,form:nonfinite,
                      subcat:Subcat,
                      sem:Sem)
        **>       (vp,form:finite,
                      subcat:NewSubcat,
                      sem:Sem)
  if add_sg3(Subcat,NewSubcat)
  morphs (X,y) becomes (X,i,e,s),
         X becomes (X,s).
```

a non-finite VP is mapped to a finite VP, provided the attachment, **add_sg3/2** succeeds in transforming the SUBCAT value to reflect agreement.

For parsing, ALE unfolds the lexicon at compile-time under application of lexical rules, with an upper bound on the depth of rule application. This was possible because lexical items were feature structures to which the code for lexical rules could apply. In the generator, however, the lexical entries themselves are compiled into pieces of code. One solution is to treat lexical rules as special unary non-chain rules, whose daughters can only have pivots corresponding to lexical entries or other lexical rules, and with bounded depth. Because the application depth is bounded, one can also unfold these lexical rule applications into the lexical entries' **non_chain_rule/4** predicates themselves. Given a lexical entry, **W ---> DescLex**, and lexical rule, **DescIn **> DescOut morphs M**, for example, we can create the clause:

```
non_chain_rule(PivotFS,RootFS,Ws,WsRest) :-
  [add DescOut to PivotFS],
  [add DescIn to LexFS],
  [add DescLex to LexFS],
  connect(PivotFS,RootFS,[MorphW|SubWs],
          SubWs,Ws,WsRest).
```

where **MorphW** is the result of applying **M** to **W**. For most grammars, this code can be heavily optimized by peephole filtering. At least part of all three descriptions needs to be enforced if there are shared structures in the input and output of the lexical rule, in order to link this to information in the lexical entry.

5 Example

An example derivation is given in Figure 1 which uses these grammar rules:

```
sent rule
  (sentence,sem:(pred:decl,args:[S])) ===>
  cat> (s,form:finite,sem:S).

s rule
  (s,form:Form,sem:S) ===>
  cat> Subj,
  sem_head> (vp,form:Form,subcat:[Subj],
             sem:S).

vp rule
  (vp,form:Form,subcat:Subcat,sem:S) ===>
  sem_head> (vp,form:Form,
             subcat:[Compl|Subcat],sem:S),
  cat> Compl.
```

The rules, **s** and **vp**, are chain rules, as evidenced by their possession of a semantic head. **sent** is a non-chain rule. Processing proceeds in alphabetical order of the labels. Arrows show the direction of control-flow between the mother and daughters of a rule. Given the input feature structure shown in (a), we obtain its semantics with **sem_select** and unify it with that of **sent**'s mother category to obtain the first pivot. **sent**'s daughter, (b), must then be recursively generated. Its semantics matches that of the lexical entry for "calls," (c), which must then be linked to (b) by chain rules. The semantic head of chain rule **vp** matches (c), to produce a mother, (d), which must be further linked, and a non-head daughter, (e), which is recursively generated by using the lexical entry for "john." A second application of **vp** matches (d), again producing a mother, (f), and a non-head daughter, (g), which is recursively generated by using the lexical entry for "up." An application of chain rule, **s**, then produces a non-head daughter, (h), and a mother. This mother is linked to (b) directly by unification.

6 Indexing

In grammars with very large lexica, generation can be considerably expensive. In the case of ALE's bottom-up parser, our interaction with the lexicon was confined simply to looking up feature structures by their phonological strings; and no matter how large the lexicon was, Prolog first argument indexing provided an adequate means of indexing by those strings. In the case of generation, we need to look up strings indexed by feature structures, which involves a much more expensive unification operation than matching strings. Given ALE's internal representation of feature structures, first argument indexing can only help us by selecting structures of the right type, which, in the case of a theory like HPSG, is no help at all, because every lexical entry is of type, *word*. (SNMP90) does not consider this problem, presumably because its data structures are much smaller.

The same problem exists in feature-based chart parsing, too, since we need to find matching feature structure chart edges given a description in a grammar rule. In the case of HPSG, this is not quite as critical given the small number of rules the theory requires. In a grammar with a large number of rules, however, a better indexing technique must be applied to chart edges as well.

The solution we adopt is to build a decision tree with features and types on the inner nodes and arcs, and code for lexical entries on the leaves. This structure can be built off-line for the entire lexicon and then traversed on-line, using a feature structure in order to avoid redundant, partially successful unification operations. Specifically, a node of the tree is labelled with a feature path in the feature structure; and the arcs emanating from a node, with the possible type values at that node's feature path.

The chief concern in building this tree is deciding which feature paths should be checked, and in which order. Our method, an admittedly preliminary one, simply indexes by all feature paths which reach into the substructure(s) identified as semantics-related by **sem_select/2**, such that shorter paths are traversed earlier, and equally short paths are traversed alphabetically. An example tree is shown in Figure 2

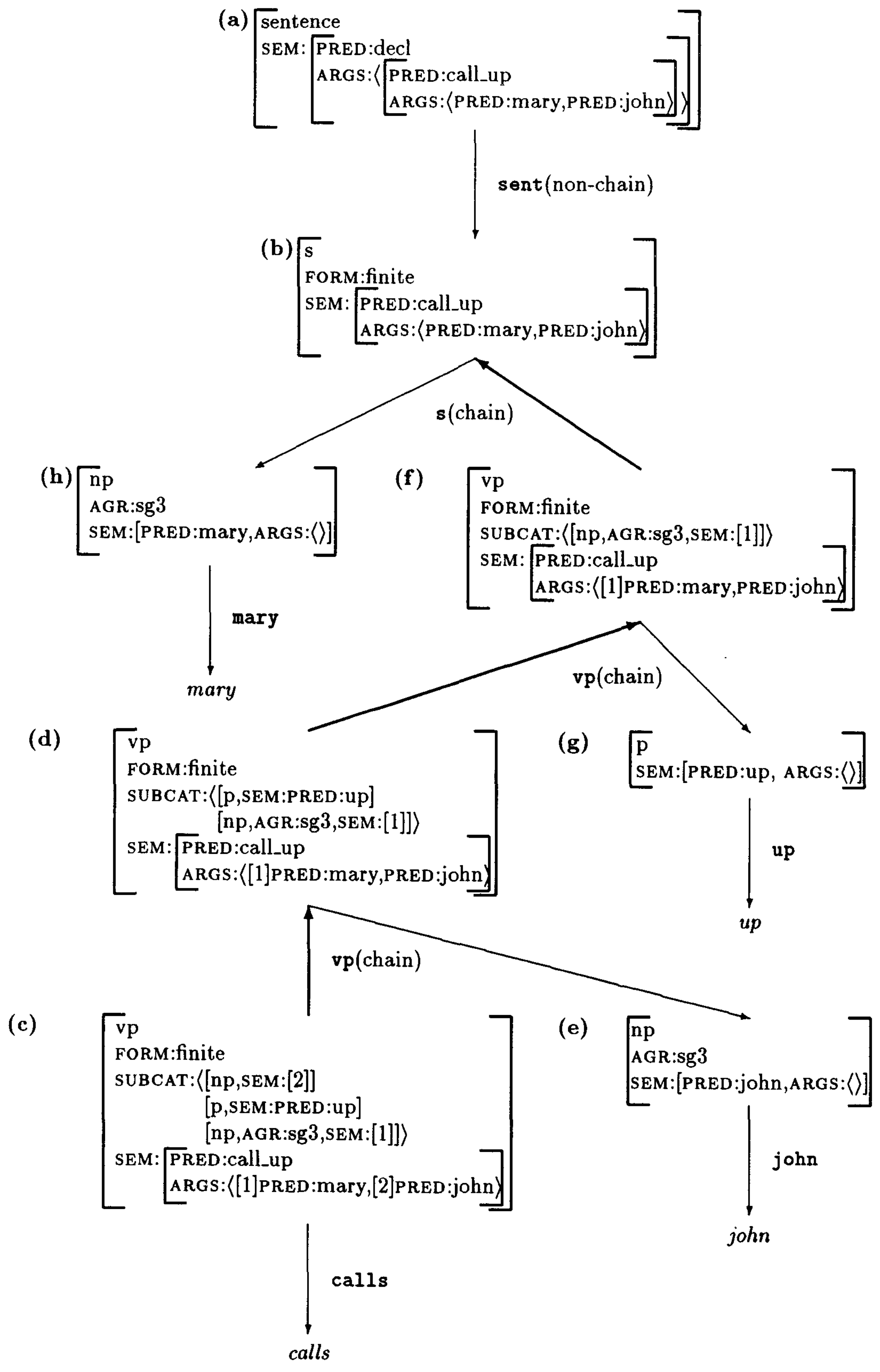

Figure 1: A Sample Generation Tree.

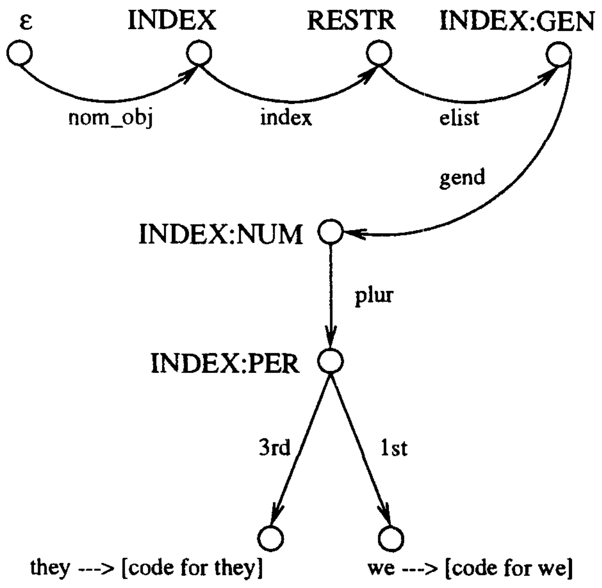

Figure 2: A sample lexical decision tree.

for the two HPSG-like lexical entries:

```
they ---> word
          ...
          CONT: nom_obj
                INDEX: index
                       GEN: gend
                       NUM: plur
                       PER: 3rd
                RESTR: elist
          ...

we   ---> word
          ...
          CONT: nom_obj
                INDEX: index
                       GEN: gend
                       NUM: plur
                       PER: 1st
                RESTR: elist
          ...
```

After the tree is built, a number is assigned to each node and the tree is compiled into a series of Prolog predicates to be used for traversal at run-time, which are then compiled by Prolog. The INDEX:PER node in Figure 2 has the following compiled code:

```
node(6,SemFS,PivotFS,RootFS,Ws,WsRest) :-
  [V := PivotFS's value at INDEX:PER],
  branch(6,V,SemFS,PivotFS,RootFS,Ws,
         WsRest).
branch(6,V,SemFS,PivotFS,RootFS,Ws,
       WsRest) :-
  [add type 3rd to V],
  node(7,SemFS,PivotFS,RootFS,Ws,WsRest).
branch(6,V,SemFS,PivotFS,RootFS,Ws,
       WsRest) :-
  [add type 1st to V],
  node(8,SemFS,PivotFS,RootFS,Ws,WsRest).

node(7,_,PivotFS,RootFS,Ws,WsRest) :-
  [add code for he to PivotFS],
  connect(PivotFS,RootFS,[he|SubWs],
          SubWs,Ws,WsRest).
node(8,_,PivotFS,RootFS,Ws,WsRest) :-
  [add code for i to PivotFS],
  connect(PivotFS,RootFS,[i|SubWs],
          SubWs,Ws,WsRest).
```

Each clause of a non-terminal node/2 finds the value of the current pivot at the current node's feature path, and then calls branch/3, which branches to a new node based on the type of that value. Leaf node clauses add the code for one of possibly many lexical entries. The non_chain_rule/4 clauses of Section 4.2 are then replaced by:

```
non_chain_rule(PivotFS,RootFS,Ws,WsRest) :-
  solve(sem_select(PivotFS,SemFS))
  node(0,SemFS,PivotFS,RootFS,Ws,WsRest).
```

As the type check on branches is made by unification, traversal of a tree can, in general, be nondeterministic. Using ALE's internal data structure for feature structures, a check to avoid infinite loops through cyclic structures during compile-time can be made in linear time.

7 Results and Future Work

Compilation of control code for head-driven generation, as outlined in Section 4, improves generation performance by a factor of about 5 on three feature-based grammars we have written and tested. The use of our indexing code independently improves generation speed by a factor of roughly 3. The combined compile-time cost for producing and compiling the control and indexing code is a factor of about 1.5. Taken as a function of maximum chain length (also declared by the user), generation is, of course, always slower with larger maxima; but performance degrades somewhat more rapidly with indexed generation than with non-indexed, and more rapidly still with compiled generation than with interpreted. In our experience, the factor of improvement decreases no worse than logarithmically with respect to maximum chain length in either case.

There are several directions in which our approach could be improved. The most important is the use of a better decision-tree growing method such as impurity-based classification ((Qui83; Utg88;

Cho91)) or concept clustering over lexical entries ((CR92)). Our current approach only guarantees that semantics-related paths are favoured over unrelated ones, and reduces redundant unifications when compared with naïve lookup in a table of feature structures. What is needed is a arrangement of nodes which minimizes the average length of traversal to a failed match, in order to prune search as soon as possible. For generation with fixed large-scale grammars, this could also involve a training phase over a corpus to refine the cost estimate based on a lexical entry's frequency. This direction is pursued further in (Pen97).

One could also explore the use of memoization for generation, to avoid regeneration of substrings, such as the "chart-based" generator of (Shi88), which was originally designed for a bottom-up generator. The best kind of memoization for a semantically driven generator would be one in which a substring could be reused at any position of the final string, possibly by indexing semantics values which could be checked for subsumption against later goals.

Another direction is the incorporation of this strategy into a typed feature-based abstract machine, such as the ones proposed in (Qu94; Win96). Abstract machines allow direct access to pointers and stack and heap structures, which can be used to make the processing outlined here even more efficient, at both compile-time and run-time. They can also be used to perform smarter incremental compilation, which is very important for large-scale grammar development. This direction is also considered in (Pen97).

8 Conclusion

We have presented the steps in compiling head-driven generation code for ALE grammar signatures, which can make use of ALE's efficient compilation of descriptions. We have also outlined a method for compiling feature-based decision trees which can be used to alleviate the lexicon indexing problem for generation, as well as the chart edge indexing problem for large-scale feature-based parsers.

All of these techniques have been implemented and will be available beginning with version 3.0 of ALE, which will be released in Spring, 1997. By compiling both logical operations and, in a processing-specific fashion, higher-level control operations, ALE can be used for very efficient, large-scale feature-based grammar design.

References

Carpenter, B., and G. Penn, 1994. The Attribute Logic Engine, User's Guide, Version 2.0.1, CMU Technical Report.

Carpenter, B., and G. Penn, 1996. Compiling Typed Attribute-Value Logic Grammars, in H. Bunt, M. Tomita (eds.), *Recent Advances in Parsing Technology*, Kluwer.

Carpineto, C. and G. Romano, 1992. GALOIS: An order-theoretic approach to conceptual clustering. *Proceedings of AAAI.*

Chou, P.A., 1991. Optimal Partitioning for Classification and Regression Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4).

Minnen, G., D. Gerdemann, and E.W. Hinrichs, 1993. Direct Automated Inversion of Logic Grammars. *Proceedings of the 4th Workshop on Natural Language Understanding and Logic Programming.*

van Noord, G., 1989. BUG: A Directed Bottom Up Generator for Unification Based Formalisms. Utrecht/Leuven working papers in Natural Language Processing 1989.

Penn, G., forthcoming. Statistical Optimization in a Feature Structure Abstract Machine. CMU Doctoral Thesis.

Popescu, O., 1996. Head-Driven Generation for Typed Feature Structures. CMU Master's Thesis.

Qu, Y., 1994. An Abstract Machine for Typed Attribute-Value Logic. CMU Master's Thesis.

Quinlan, J., 1983. Learning Efficient Classification Procedures. In Michalski, Carbonell, Mitchell (eds.), *Machine Learning: an artificial intelligence approach*, Morgan Kaufmann.

Shieber, S.M., 1988. A Uniform Architecture for Parsing and Generation. *Proceedings of the 12th International Conference on Computational Linguistics*, pp. 614–619.

Shieber, S.M., G. van Noord, R.C. Moore and F.C.N. Pereira, 1990. Semantic-head-driven Generation. *Computational Linguistics*, 16.

Strzalkowski, T., 1990. Reversible Logic Grammars for Natural Language Parsing and Generation. *Canadian Computational Intelligence Journal*, 6(3), pp. 145–171.

Utgoff, 1988. ID5: an incremental ID3. *International Machine Learning Conference*, Ann-Arbor.

Wintner, S., 1996. An Abstract Machine for Unification Grammars. Technion Doctoral Thesis.