ALE

The Attribute Logic Engine

User's Guide

Version 3.2.1
December 2001

Bob Carpenter
SpeechWorks Research
55 Broad St.
New York, NY 10004
USA

carp@colloquial.com

Gerald Penn
Department of Computer Science
University of Toronto
10 King's College Rd.
Toronto M5S 3G4
Canada
gpenn@cs.toronto.edu

Contents

Preface – Version 3.0

ALE 3.0 is completely compatible with ALE 2.0 grammars, and adds the following new features:

- A semantic-head-driven generator, based on the algorithm presented in Shieber et al. (1990). The generator was adapted to the logic of typed feature structures by Octav Popescu in his Carnegie Mellon Master's Thesis, Popescu (1996). Octav also wrote most of the generation code for this release. Grammars can be compiled for parsing only, generation only, or both. Some glue-code is also available from the ALE homepage, to parse and generate with different grammars through a unix pipe.

- A source-level debugger with a graphical XEmacs interface. This debugger works only with SICStus Prolog 3.0.6 and higher. A debugger with reduced functionality will be made available to SWI Prolog users in a later release. This debugger builds on, and incorporates the functionality of the code for the SICStus source-level debugger, written by Per Mildner at Uppsala University.

- Functional descriptions. Instead of binding variables in a description and calling a procedural attachment, e.g., `a cons f:X,g:Y,h:Z goal append(X,Y,Z)`, it is now possible to incorporate certain functional relations into descriptions themselves, e.g., `a cons f:X,g:Y,h:append(X,Y)`.

- `a_/1` atoms. There are now an infinite number of atoms (types with no appropriate features), implicitly declared in every signature. These atoms can be arbitrary Prolog terms, including unbound variables, and can be used wherever normal ALE types can, e.g., `f:(a_ p(3.7))`. `a_/1` atoms are extensional as Prolog terms, i.e., are taken to be identical according to the Prolog predicate, `==/2`. In particular, this means that ground atoms behave exactly as ALE extensional types.

- Optional edge subsumption checking. For completeness of parsing, one only needs to ensure that, for every pair of nodes in the chart, the most general feature structure spanning those nodes is stored in the chart. This can reduce the number of edges in many domains.

- An autonomous `intro/2` operator. Features can now be declared on their own in a separate part of the grammar.

- Default specifications for types. These are NOT default types. If a type appears on the right-hand side of a `sub/2` or `intro/2` specification, but not on

the left-hand side of one, ALE will assume this type is maximal, i.e., assume the specification, *Type* `sub []`. Similarly, if it occurs on a left-hand side, but not on a right-hand side, ALE will assume the type is immediately subsumed by `bot`, the most general type. In both cases, ALE will announce these assumptions during compilation.

- Several bug corrections and more compile-time warning and error messages.
- An SWI Prolog 2.9.7 port. Version 3.0 will be the last version for which we will port the system to Quintus Prolog. We will now support ALE for SICStus Prolog and SWI Prolog. The SICStus version of the system works for SICStus Prolog 3.0.5 and higher, except for the source-level debugger, which requires version 3.0.6.

We would like to thank all of the users who have supplied us with feedback and suggestions over the past few years for how to improve ALE. In particular, we would like to thank Ion Androutsopoulos, Mike Calcagno, Mats Carlsson, Frederik Fouvry, Gertjan van Noord, Peter van Roy, Margriet Verlinden, and Jan Wielemaker for their patient assistance. This release incorporates many, but unfortunately not all of those changes. Quite a few more will be made in the next several releases, along with many performance improvements.

Bob Carpenter and Gerald Penn
Murray Hill and Tuebingen, March 1998

Preface – Version 2.0

ALE 2.0 is a proper extension of version 1.0. Specifically, version 2.0 will run any grammar that will run under version 1.0. But version 2.0 includes many extensions to version 1.0, including the following.

- Inequations
- Extensionality
- General Constraints on Types
- Mini-interpreter
- Error-suppression

Inequations allow inequational constraints to be imposed between two structures. Extensionality allows structures of specified types to be identified if they are structurally identicial. Together, these provide the ability to simulate Prolog II programs (Colmerauer 1987). ALE 2.0 also allows general constraints to be placed on types, using arbitrary descriptions from the constraint language, including path equations, inequations and disjunctions, and procedural attachments. It also has a mini-interpreter, which allows the user to traverse and edit an ALE parse tree. Error messages for incompatible descriptions are now automatically disabled during lexicon and empty category compilation.

The second release of ALE, Version 2.0, is based on an extension of the first version of ALE, that was completed for Gerald Penn's (1993) MS Project in the Computational Linguistics Program at Carnegie Mellon University.

There are many people whom we would like to thank for their comments and feedback on version 1.0 and β-versions of 2.0. These people have actually used the system in their research and have thus had the best opportunity to provide us with practical feedback. First, we would like to thank the first group of users, housed at Sharp Laboratories of Europe, located in Oxford, England, including Pete Whitelock, Antonio Sanfillipo, and Osamu Nishida. They not only used the system but provided feedback on the code. Secondly, the group at University of Tübingen, who are developing a competing system, Troll, have rigorously tested existing systems, including ALE, both for their ability to express grammars naturally and for efficiency. Specifically, we would like to thank Detmar Meurers, Dale Gerdemann, Thilo Götz, Paul King, John Griffith, and Erhard Hinrichs. John and Thilo also provided the changes necessary for the system to run directly in Quintus Prolog. This group is undoubtedly the best informed when it comes to implemented grammar formalisms. We would also like to thank the grammar development group at

Stanford University, including Ivan Sag, Chris Manning, Suzanne Riehemann. We would further like to thank Bob Kasper, Carl Pollard, and Andreas Kathol of the Ohio State University, for a great deal of feedback on the design of HPSG grammars in general, and ALE implementations of them in particular. Chris Manning, in addition, found a bug in SICStus Prologs prior to 2.1.8, which prevented cyclic structures from being used in completed chart edges, a bug found by both Steven Bird of Edinburgh and C. J. Rupp of IDSIA. Their feedback on Bob Carpenter's prototype implementation of HPSG for English led to the design of Gerald Penn's much more comprehensive implementation of HPSG and was the primary impetus for the importation of general type constraints into version 2.0. Next, we would like to thank Claire Gardent, who has been using ALE to develop discourse grammars in Amsterdam. We should also thank Carsten Guenther and Markus Walther, of the Universities of Hamburg and Düsseldorf, respectively, who have used the system to develop phonological grammars. Finally, we should thank Michael Mastroianni, who implemented a comprehensive approach to constraint-based phonology in ALE (Mastroianni 1993). He suffered through early, buggy versions of the system, thus sparing the rest of us much of that pain. The feedback we received from these users was invaluable.

We would like to thank EAGLES, the European Advisory Group on Linguistic Engineering Standards, for allowing us to present our system at a meeting in Saarbrücken in March 1993 of the European Expert Group on Linguistic Formalisms devoted to implemented formalisms. We learned a great deal from the other participants in the workshop including especially Jochen Dörre, Michael Dorna, and Martin Emele, of Stuttgart, and Andreas Podelski, then associated with the Digital Equipment Paris Research Lab. We also benefitted from discussions with Hans Uszkoreit, Rolf Backofen, and Uli Krieger, of Saarbrücken, Steve Pulman from SRI in Cambridge, and C. J. Rupp and Graham Russell, of ISSCO in Switzerland.

We had many discussions of the ALE formalism at the HPSG workshop running concurrently with the LSA Linguistic Institute in Columbus. We would especially like to thank Gregor Erbach for comments on our system, including benchmark test results. We would also like to thank Hiroshi Tusda, of the Institute for New Gernateion Computer Technology, for discussion of our systems and comparisons to his system, cu-Prolog. We also discussed ALE heavily during the workshop on implementations of attribute-value logics, during the 1993 Summer School on Logic, Language, and Information in Lisbon, Portugal. We especially benefitted from discussions with Suresh Manandhar, of the University of Edinburgh, and Gerrit Rentier of Tilburg University, and Gert Webelhuth of the University of North Carolina, among those we have not already thanked. We also benefitted from discussions with Ed Stabler and Mark Johnson, and from sitting in on their class on the implementation of constraint-based grammars.

We would also like to thank Ann Copestake and Ted Briscoe, of the Cambridge Computing Laboratory, for feedback on the design of the system.

We would like to thank Richard O'Keefe, who provided some invaluable feedback on coding style. Of course, any glitches or failure to follow his excellent example are our own.

We would also like to thank Elizabeth Hinkelman, who runs the Software Registry, and Mark Kantrowitz, who administers the Prolog Resource Guide and the Prime Time Freeware for AI CD-ROM. They have helped in publicizing the system

description as well as providing access.

The extensions we have not made, though would like to, include the addition of:

- Primitive, Atomic Data Types
- Parametric Types
- Partial Type Inference
- Assert Mode in Compiler
- Peephole Code Optimization
- Subsumption Checking of Chart Edges

The incorporation of Prolog data types such as Real, Integer, Character, and String, is straightforward theoretically, but not so straightforward in terms of ALE. The same goes for parametric polymorphism at the type level. Partial type inference could provide a great deal of optimization in some circumstances. We could not figure out how to incoprorate these three changes without drastically modifying the underlying representations and algorithms. The remaining changes are lying dormant because we have other obligations.

The next wave of development of attribute-logic grammars should not be in Prolog, but rather through the use of a direct abstract machine. Bob Carpenter has worked on an abstract machine with Yan Qu, in the context of her MS project in the Carnegie Mellon Computational Linguistics Program, and with Shuly Wintner, of the Technion, in Haifa, Israel, who is writing a PhD dissertation on the topic. Such an undertaking is also underway among the LIFE community, led by Hassan Aït-Kaci, Andreas Podelski, and Peter van Roy.

We would like to thank a number of people for discovering bugs and providing comments on Version 2.0: Ingo Schroeder, Frank Morawietz, Detmar Meurers, Rob Malouf, Frederik Fouvry, Jo Calder, and Suresh Manandhar.

Finally, we would like to thank Jo Calder,Chris Brew, Kevin Humphreys, and Mike Reape, who developed the Pleuk grammar development environment as well as interfacing it to ALE. Details of that system can be found in the appropriate Appendix.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship (for Gerald Penn). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Bob Carpenter and Gerald Penn
Pittsburgh, August 1994

Preface – Version Beta

A number of people have asked me to make this system, along with its documentation, available to the public. Now that it's available, I hope that it's useful. But a word of caution is in order. The system is still only a prototype, hence the label "version β."

Any bug reports would be greatly appreciated. But what I'd really like is comments on the functionality of the system, as well as on the utility of its documentation. I am also interested in hearing of any applications that are made of the system. I would also be glad to answer questions about the system. I have tried to document the strategies used by ALE in this guide. I have also tried to comment the code to the point where it might be adaptable by others. I would, of course, be interested in any kind of improvements or extensions that are discovered or developed, and would like to have the chance to incorporate any such improvements in future versions of this package.

In the implementation, I have endeavored to follow the logic programming methodology laid out by O'Keefe (1990), but there are many spots where I have fallen short. Thus the code is not as fast as it could be, even in Prolog. But I view this system more as a prototype, indicating the utility of a typed logic programming and grammar development system. Borrowing techniques from the WAM directly, implementing an abstract machine C, would lead to roughly a 100-fold speedup, as there is no reason that ALE should be slower than Prolog itself.

I would like to acknowledge the help of Gerald Penn in working through many implementation details of a general constraint resolver, which was the inspiration for this implementation. This version of the system is a great improvement on the last version due to Gerald's work on the system. Secondly, I would like to thank Michael Mastroianni, who has actually used the system to develop grammars for phonology. Finally, I would like to thank Carl Pollard and Bob Kasper for looking over a grammar of HPSG coded in ALE and providing the impetus for the inclusion of empty categories and lexical rules.

The system is available without charge from the author. It is designed to run in either SICStus or Quintus Prologs.

Bob Carpenter
Pittsburgh, 1992

Chapter 1

Introduction

This report serves as an introduction to both the ALE formalism and its Prolog implementation. ALE is an integrated phrase structure parsing and definite clause logic programming system in which the terms are typed feature structures. Typed feature structures combine type inheritance and appropriateness specifications for features and their values. The feature structures used in ALE generalize the common feature structure systems found in the linguistic programming systems PATR-II and FUG, the grammar formalisms HPSG and LFG, as well as the logic programming systems Prolog-II and LOGIN. Programs in any of these languages can be encoded directly in ALE.

Terms in grammars and logic programs are specified in ALE using a typed version of Rounds and Kasper's attribute-value logic with variables. At the term level, we have variables, types, feature value restrictions, path equations, inequations, general constraints, and disjunction. The definite clause programs allow disjunction, negation and cut, specified with Prolog syntax. Phrase structure grammars are specified in a manner similar to DCGs, allowing definite clause procedural attachment. The grammar formalism also fully supports empty categories. Lexical development is supported by a very general form of lexical rule which operates on both categories and surface strings. Macros are available to help organize large descriptions, either in programs or in grammars. Both definite clause programs and grammars are compiled into abstract machine instructions. These instructions are then interpreted by an emulator compiled from the type specifications. Like Prolog compilers, a structure copying strategy is used for matching both definite clauses and grammar rules.

For parsing, ALE compiles from the grammar specification a Prolog-optimized bottom-up, dynamic chart parser. Definite clauses are also compiled into Prolog. As it stands, the current version of ALE, running the feature-structure-based naive reverse of a 30-element list on top of the SICStus 3.7 native code compiler, runs at roughly 152,300 logical inferences per second (LIPS) on a Sun Ultra Enterprise 450. This is slightly faster than the speed of the SICStus 3.7 interpreter on the Prolog naive reverse, 60% as fast as SWI Prolog 3.2.7, and about 9% as fast as the SICStus 3.7 compact-code compiler. The definite clause compiler performs last call optimization, but does not index first arguments or use specialised list cells at the WAM level. Thus it will perform relatively well versus non-optimized interpreters, but lag further behind compiled grammars when programs are written in a more

sophisticated manner than naive reverse.

Full details of the theory behind ALE can be found in Carpenter (1992).

The user who is only interested in definite clause programming can skip the material on phrase structure grammars, while those interested in only grammars without procedural attachments may skip the material in the section on definite clauses.

Chapter 2

Prolog Preliminaries

While it is not absolutely necessary, some familiarity with logic programming in general, and Prolog in particular, is helpful in understanding the definite clause portion of ALE. Similarly, experience with unification grammar systems such as PATR-II, DCGs, or FUG is helpful in understanding the phrase structure component of the system. In particular, writing efficient programs and grammars in ALE involves the same kinds of strategies necessary for writing efficient programs in Prolog or PATR-II. For those not familiar with Prolog, the sequence of two books by Sterling and Shapiro (1986) and by O'Keefe (1990) are excellent general introductions to the theory and practice of logic programming. For those not familiar with unification-based grammar formalisms, Shieber (1986), Gazdar and Mellish (1987) and Pereira and Shieber (1987) are useful resources.

For those not familiar with Prolog, we need to point out the salient features of the language which will be assumed throughout this report. This section contains all of the information necessary about Prolog required to run ALE.

2.1 Terms

A Prolog *constant* is composed of either a sequence of characters and/or underscores, beginning with a lower case letter, a number, or any sequence of symbols surrounded by apostrophes. So, `abc, johnDoe, b_17, 123, 'JohnDoe', '65$'`, and `'_65a.'` are constants, but `A19, JohnDoe, B_112, _au8`, and `[dd,e]` are not (although see the warning at the end of this section). A variable, on the other hand, is any string of letters, underscores or numbers beginning with a capital letter. Thus `C, C_foo,` and `TR5ab` are variables, but `1Xa, aXX`, and `_Xy`[1] are not.

In general, it is a bad idea to have constants or variables which are only distinguished by the capitalization of some of their letters. For instance, while `aBa` and `aba` are different constants, they should not both be used in one program. One reason for this in the context of ALE is that the output routines adopt standard capitalization conventions which hide the differences between such constants.

Warning: As pointed out to us by Ingo Schroeder, constants or atoms beginning with a capital letter are not treated properly by the compiler. Thus constants such

[1] Technically, a variable may begin with an underscore, but such variables, said to be *anonymous*, have a very different status than those which begin with a capital letter. The use of anonymous variables is discussed later.

as `'Foo'` should not be used.

2.2 Space and Comments

In your own program and grammar files, extra *whitespace* between symbols beyond that needed to separate constants or variables is ignored. Whitespace consists of either spaces, blank lines or line breaks are ignored. This allows you to format your programs in a manner that is readable. Furthermore, any symbols on a line appearing after a `%` symbol are treated as *comments* and ignored.

2.3 Running Prolog

To fire up Prolog locally, you should contact your systems administrator. You should have either SICStus or SWI Prolog, or a Prolog compiler compatible with one of these. Once Prolog is fired up, you will see a *prompt*. The Prolog prompt should look like:

```
| ?-
```

It is important that Prolog be invoked from a directory for which the user has write permission. ALE, in the process of compiling user programs, writes a number of local files.

2.4 Queries

What you type after the prompt is called a *query*. Queries should always end with a period and be followed by a carriage return. In fact, all of the grammar rules, definite clauses, macros and lexical entries in your programs should also end with periods. Most of the interface in ALE is handled directly by top-level Prolog queries. Many of these will return `yes` or `no` after they are called, the significance of which within ALE is explained on a query by query basis.

2.5 Running ALE

To run ALE, it is only necessary to type the following query:

```
| ?- compile(File).
```

where *File* is the file in which the file `ale.pl` resides. SWI Prolog users must type:

```
| ?- consult(File).
```

Note that *File* does not have to be local to the directory from which Prolog was invoked.

In SICStus Prolog, when ALE is loaded, it turns on Prolog character escapes. ALE will not be able to generate code properly during compilation without this.

2.6 Exiting Prolog and Breaking

To exit from Prolog, you can type `halt` at any prompt (followed by a period, of course).

If you find Prolog hanging at some point, and you are working on a standard Unix implementation, typing `control-c` should produce something like the following message:

```
Prolog interruption (h for help)?
```

You should reply with the character `a`, with or without a following period, followed by a carriage return. If this doesn't work, typing `control-z` should take you out of Prolog altogether.

2.7 Saved States

All information concerning an ALE state is encoded in the current Prolog state. Thus, any options presented by the local system to save Prolog states should be able to save ALE states.

Chapter 3

Feature Structures, Types and Descriptions

This section reviews the basic material from Carpenter (1992), Chapters 1–10, which is necessary to use ALE.

3.1 Inheritance Hierarchies

ALE is a language with strong typing. What this means is that every structure it uses comes with a type. These types are arranged in an inheritance hierarchy, whereby type constraints on more general types are inherited by their more specific subtypes, leading to what is known as *inheritance-based polymorphism*. Inheritance-based polymorphism is a cornerstone of object-oriented programming. In this section, we discuss the organization of types into an inheritance hierarchy. Thus many types will have *subtypes*, which are more specific instances of the type. For instance, `person` might have subtypes `male` and `female`.

ALE does much of its processing of types at compile time, as it is reading and processing the grammar file. Thus the user is required to declare all of the types that will be used along with the subtyping relationship between them. An example of a simple ALE type declaration is as follows:

```
bot sub [b,c].          % two basic types -- b and c
  b sub [d,e].
    d sub [g,h].
    e sub [].
  c sub [d,f].          % b and c unify to d
    f sub [].
```

There are quite a few things to note about this declaration. The types declared here are `bot, b, c, d, e, f` and `g`. Note that each type that is mentioned gets its own specification. Of course, the whitespace is not important, but it is convenient to have each type start its own line. A simple type specification consists of the name of the type, followed by the keyword `sub`, followed by a list of its subtypes (separated by whitespace). In this case, `bot` has two subtypes, `b` and `c`, while `f, d` and `e` have no subtypes. The subtypes are specified by a Prolog list. In this case, a Prolog list consists of a sequence of elements separated by commas and enclosed in

square brackets. Note that no whitespace is needed between the list brackets and types, between the types and commas, or between the final bracket and the period. Whitespace is only needed between constants. The extra whitespace on successive lines is conventional, indicating the level in the ordering for the user, but is ignored by the program. Also notice that there are comments on two of the lines; recall that comments begin with a `%` sign and continue the length of the line. Every type (except `a_/1` atoms, discussed below) must have at most one `sub` declaration, i.e., all of the immediate subtypes must be declared in one declaration.

The subtyping relation is only specified by *immediate* subtyping declarations; but subtyping itself is transitive. Thus, in the example, `d` is a subtype of `c`, and `c` is a subtype of `bot`, so `d` is also a subtype of `bot`. The user only needs to specify the direct subtyping relationship. The transitive closure of this relation is computed by the compiler. While redundant specifications, such as putting `d` directly on the subtype list of `bot`, will not alter the behavior of the compiler, they are confusing to the reader of the program and should be avoided. In addition, the derived transitive subtyping relationship must be anti-symmetric. In particular, this means that there should not be two distinct types each of which is a subtype of the other.

There are two additional restrictions on the inheritance hierarchy beyond the requirement that it form a partial order. First, there is a special type `bot`, which must be declared as the unique most general type. In other words, every type must be a subtype of `bot`. If a type is used on the left-hand side of a `sub` declaration, but never declared as a sub-type of anything else, it is assumed that this type is an immediate subtype of `bot`. Similarly, ALE assumes that all types for which no subtypes are declared are maximal, i.e., have no subtypes.

The second and more subtle restriction on type hierarchies is that they be *bounded complete*. Since type declarations must be finite, this amounts to the restriction that every pair of types which have a common subtype have a unique most general common subtype. In the case at hand, `b` and `c` have three common subtypes, `d, g`, and `h`. But these subtypes of `b` and `c` are ordered in such a way that `d` is the most general type in the set, as both `g` and `h` are subtypes of `d`. An example of a type declaration violating this condition is:

```
bot sub [a,b].
  a sub [c,d].
    c sub [].
    d sub [].
  b sub [c,d].
```

The problem here is that while `a` and `b` have two common subtypes, namely `c` and `d`, they do not have a most general common subtype, since `c` is not a subtype of `d`, and `d` is not a subtype of `c`. In general, a violation of the bounded completeness condition such as is found in this example can be patched without destroying the ordering by simply adding additional types. In this case, the following type hierarchy preserves all of the subtyping relations of the one above, but satisfies bounded completeness:

```
bot sub [a,b].
  a sub [e].
    e sub [c,d].
      c sub [].
```

```
    d sub [].
b sub [e].
```

In this case, the new type `e` is the most general subtype of `a` and `b`.

This last example brings up another point about inheritance hierarchies. When a type only has one subtype, the system provides a warning message (as opposed to an error message). This condition will not cause any compile-time or run-time errors, and is perfectly compatible with the logic of the system. It is simply not a very good idea from either a conceptual or implementational point of view. For more on this topic, see Carpenter (1992:Chapter 9).

3.2 Feature Structures

The primary representational device in ALE is the *typed feature structure*. In phrase structure grammars, feature structures model categories, while in the definite clause programs, they serve the same role as first-order terms in Prolog, that of a universal data structure. Feature structures are much like the frames of AI systems, the records of imperative programming languages like C or Pascal, and the feature descriptions used in standard linguistic theories of phonology, and more recently, of syntax.

Rather than presenting a formal definition of feature structures, which can be found in Carpenter (1992:Chapter 2), we present an informal description here. In fact, we begin by discussing feature structures which are not necessarily well-typed. In the next section, the type system is presented.

A feature structure consists of two pieces of information. The first is a type. Every feature structure must have a type drawn from the inheritance hierarchy. The other kind of information specified by a feature structure is a finite, possibly empty, collection of feature/value pairs. A feature value pair consists of a feature and a value, where the value is itself a feature structure. The difference between feature structures and the representations used in phonology and in GPSG, for instance, is that it is possible for two different substructures (values of features at some level of nesting) to be token identical in a feature structure. Consider the following feature structure drawn from the lexical entry for *John* in the categorial grammar in the appendix, displayed in the output notation of ALE:

```
cat
QSTORE e_list
SYNSEM basic
       SEM j
       SYN np
```

The type of this feature structure is `cat`, which is interpreted to mean it is a category. It is defined for two features, `QSTORE` and `SYNSEM`. As can be seen from this example, we follow the HPSG notational convention of displaying features in all caps, while types are displayed in lower case. Also note that features and their values are printed in alphabetic order of the feature names. In this case, the value of the `QSTORE` feature is the simple feature structure of type `e_list`,[1] which has no feature values. On the

[1]Set values, like those employed in HPSG, are not supported by ALE. In the categorial grammar

other hand, the feature SYNSEM has a complex feature as its value, which is of type basic, and has two feature values SEM and SYN, both of which have simple values.

This last feature structure doesn't involve any structure sharing. But consider the lexical entry for *runs*:

```
cat
QSTORE e_list
SYNSEM backward
       ARG basic
           SEM [0] individual
           SYN np
       RES basic
           SEM run
               RUNNER [0]
           SYN s
```

Here there is structure sharing between the path SYNSEM ARG SEM and the path SYNSEM RES SEM RUNNER, where a *path* is simply a sequence of features. This structure sharing is indicated by the *tag* [0]. In this case, the sharing indicates that the semantics of the argument of *runs* fills the runner role in the semantics of the result. Also note that a shared structure is only displayed once; later occurrences simply list the tag. Of course, this example only involves structure sharing of a very simple feature structure, in this case one consisting of only a type with no features. In general, structures of arbitrary complexity may be shared, as we will see in the next example.

ALE, like Prolog II and HPSG, but unlike most other systems, allows cyclic structures to be processed and even printed. For instance, consider the following representation we might use for the liar sentence *This sentence is false*:

```
[0] false
    ARG1 [0]
```

In this case, the empty path and the feature ARG1 share a value. Similarly, the path ARG1 ARG1 ARG1 and the path ARG1 ARG1, both of which are defined, are also identical. But consider a representation for the negation of the liar sentence, *It is false that this sentence is false*:

```
false
ARG1 [0] false
     ARG1 [0]
```

Unlike Prolog II, ALE does not necessarily treat these two feature structures as being identical, as it does not conflate a cyclic structure with its infinite unfolding. We take up the notion of token identical structures in the section below on extensionality.

It is interesting to note that with typed feature structures, there is a choice between representing information using a type and representing the same information using feature values. This is a familiar situation found in most inheritance-based representation schemes. Thus the relation specified in the value of the path SYNSEM RES SEM is represented using a type, in:

in the appendix, they are represented by lists and treated by attached procedures for union and selection.

```
SEM run
    RUNNER [0]
```

An alternative encoding, which is not without merit, is:

```
SEM unary_rel
    REL run
    ARG1 [0]
```

In general, type information is processed much more efficiently than feature value information, so as much information as possible should be placed in the types. The drawback is that type information must be computed at compile-time and remain accessible at run-time. More types simply require more memory.[2]

3.3 Subsumption and Unification

Feature structures are inherently partial in the information they provide. Based on the type inheritance ordering, we can order feature structures based on how much information they provide. This ordering is referred to as the *subsumption* ordering. The notion of subsumption, or information containment, can be used to define the notion of unification, or information combination. Unification conjoins the information in two feature structures into a single result if they are consistent and detects an inconsistency otherwise.

3.3.1 Subsumption

We define subsumption, saying that F *subsumes* G, if and only if:

- the type of F is more general than the type of G
- if a feature f is defined in F then f is also defined in G such that the value in F subsumes the value in G
- if two paths are shared in F then they are also shared in G

Consider the following examples of subsumption, where we let < stand for subsumption:

```
agr         <  agr
PERS first     PERS first
               NUM plu

sign               phrase
SUBJ agr        <  SUBJ agr
     PERS pers          PERS first
                        NUM plu
```

[2] In general, the amount of memory required to represent n types is proportional to the number of pairs of consistent types. In the worst case, this is $\mathcal{O}(n^2)$ in the number of types.

```
sign                sign
SUBJ agr            SUBJ [0] agr
     PERS first          PERS first
     NUM plu     <       NUM plu
OBJ agr             OBJ [0]
    PERS first
    NUM plu

false               false              [1] false
ARG1 false      <   ARG1 [0] false  <  ARG1 [1]
     ARG1 false         ARG1 [0]
```

Note that the second of these subsumptions holds only if `pers` is a more general type than `first`, and `sign` is a more general type than `phrase`. It is also important to note that the feature structure consisting simply of the type `bot` will subsume every other structure, as the type `bot` is assumed to be more general than every other type.

3.3.2 Unification

Unification is an operation defined over pairs of feature structures that combines the information contained in both of them if they are consistent and fails otherwise. In ALE, unification is very efficient.[3] Declaratively, unifying two feature structures computes a result which is the most general feature structure subsumed by both input structures. But the operational definition is more enlightening, and can be given by simple conditions which tell us how to unify two structures. We begin by unifying the types of the structures in the type hierarchy. This is why we required the bounded completeness condition on our inheritance hierarchies; we want unification to produce a unique result. If the types are inconsistent, unification fails. If the types are consistent, the resulting type is the unification of the input types. Next, we recursively unify all of the feature values of the structures being unified which occur in both structures. If a feature only occurs in one structure, we copy it over into the result. This algorithm terminates because we only need to unify structures which are non-distinct and there are a finite number of nodes in any input structure.

Some examples of unification follow, where we use + to represent the operation:

```
agr          +  agr      =  agr
PERS first      NUM plu     PERS first
                            NUM sing

sign              sign             sign
SUBJ agr       +  SUBJ [0] bot  =  SUBJ [0] agr
     PERS 1st     OBJ [0]              PERS first
```

[3] Using a typed version of the Martelli and Montanari (1982) algorithm, which was adapted to cyclic structures by Jaffar (1984), unification can be performed in what is known as quasi-linear time in the size of the input structures, where in this case, quasi-linear in n is defined to be $\mathcal{O}(n \cdot ack^{-1}(n))$, where ack^{-1} is the inverse of Ackermann's function, which will never exceed 4 or 5 for structures that can be represented on existing computers. There is also a factor in the complexity of unification stemming from the type hierarchy and appropriateness conditions, which we discuss below.

```
OBJ agr                                  NUM plu
    NUM plu                         OBJ [0]

t           t           t
F [0] t  +  F t      =  F [1] t
G [0]         F [1]       F [1]
              G [1]       G [1]

agr         +  agr          =  *failure*
PERS first     PERS second

e_list  +  ne_list     =  *failure*
           HD a
           TL e_list
```

Note that the second example respects our assumption that the type `bot` is the most general type, and thus more general than `agr`. The second example illustrates what happens in a simple case of structure sharing: information is retrieved from both the `SUBJ` and `OBJ` and shared in the result. The third example shows how two structures without cycles can be unified to produce a structure with a cycle. Just as the feature structure `bot` subsumes every other structure, it is also the identity with respect to unification; unifying the feature structure consisting just of the type `bot` with any feature structure F results simply in F. The last two unification attempts fail, assuming that the types `first` and `second` and the types `e_list` and `ne_list` are incompatible.

3.4 Inequations

Feature structures may also incorporate inequational constraints following (Carpenter 1992), which is in turn based on the notion of inequation in Prolog II (Colmerauer 1987). For instance, we might have the following representation of the semantics of a sentence:

```
SEM binary_rel
    REL know
    ARG1  [0] referent
          GENDER masc
          PERS third
          NUM sing
    ARG2  [1] referent
          GENDER masc
          PERS third
          NUM sing
[0] =\= [1]
```

Below the feature information, we have included the constraint that the value of the structure `[0]` is not identical to that of structure `[1]`. As a result, we cannot unify this structure with the following one:

```
REL know
ARG1 [2]
ARG2 [2]
```

Any attempt to unify the structures `[0]` and `[1]` causes failure.

3.5 Type System

As we mentioned in the introduction, what distinguishes ALE from other approaches to feature structures and most other approaches to terms, is that there is a strong type discipline enforced on feature structures. We have already demonstrated how to define a type hierarchy, but that is only half the story with respect to typing. The other component of our type system is a notion of feature *appropriateness*, whereby each type must specify which features it can be defined for, and furthermore, which types of values such features can take. The notion of appropriateness used here is similar to that found in object-oriented approaches to typing. For instance, if a feature is appropriate for a type, it will also be appropriate for all of the subtypes of that type. In other words, appropriateness specifications are inherited by a type from its supertypes. Furthermore, value restrictions on feature values are also inherited. Another important consideration for ALE's type system is the notion of type inference, whereby types for structures which are underspecified can be automatically inferred. This is a property our system shares with the functional language ML, though our notion of typing is only first-order. To further put ALE's type system in perspective, we note that type inheritance must be declared by the user at compile time, rather than being inferred. Furthermore, types in ALE are semantic, in Smolka's (1988b) terms, meaning that types are used at run-time. Even though ALE employs semantic typing, the type system is employed statically (at compile-time) to detect type errors in grammars and programs.

As an example of an appropriateness declaration, consider the simple type specification for lists with a head/tail encoding:

```
bot sub [list,atom].
  list sub [e_list,ne_list].
    e_list sub [].
    ne_list sub []
            intro [hd:bot,
                   tl:list].
atom sub [a,b].
  a sub [].
  b sub [].
```

This specification tells us that a list can be either empty (`e_list`) or non-empty (`ne_list`). It implicitly tells us that an empty list cannot have any features defined for it, since none are declared directly or inherited from more general types. The declaration also tells us that a non-empty list has two features, representing the head and the tail of a list, and, furthermore, that the head of a list can be anything (since every structure is of type `bot`), but the tail of the list must itself be a list. Note that features must also be Prolog constants, even though the output routines convert them to all caps. The appropriateness declaration, `intro`, can be specified

along with subsumption, as shown above, or separately; but for any given type, all features must be declared at once. If no `intro` declaration is given for a type, it is assumed that that type introduces no appropriate features. If an `intro` declaration is made for a type that does not occur on either side of a `sub` declaration, that type is assumed to be an immediate subtype of `bot` with no subtypes of its own. If a value restrictor (such as `list` above for feature `tl`) does not occur on either side of a `sub` declaration, it too is assumed to be maximal and an immediate subtype of `bot`.

In ALE, every feature structure must respect the appropriateness restrictions in the type declarations. This amounts to two restrictions. First, if a feature is defined for a feature structure of a given type, then that type must be appropriate for the feature. Furthermore, the value of the feature must be of the appropriate type, as declared in the appropriateness conditions. The second condition goes the other way around: if a feature is appropriate for a type, then every feature structure of that type must have a value for the feature. A feature structure respecting these two conditions is said to be *totally well-typed* in the terminology of Carpenter (1992, Chapter 6).[4] For instance, consider the following feature structures:

```
list
HD a
TL bot

ne_list
HD bot
TL ne_list
   HD atom
   TL list

ne_list
HD [0] ne_list
   HD [0]
   TL [0]
TL e_list
```

The first structure violates the typing condition because the type `list` is not appropriate for any features, only `ne_list` is. But even if we were to change its type to `ne_list`, it would still violate the type conditions, because `bot` is not an appropriate type for the value of `TL` in a `ne_list`. On the other hand, the second and third structures above are totally well-typed. Note that the second such structure does not specify what kind of list occurs at the path `TL TL`, nor does it specify what the `HD` value is, but it does specify that the second element of the list, the `TL HD` value is an `atom`, but it doesn't specify which one.

To demonstrate how inheritance works in a simple case, consider the specification fragment from the categorial grammar in the appendix:

[4]The choice of totally well-typed structures was motivated by the desire to represent feature structures as records at run-time, without listing their features. Internally, a feature structure is represented as a term of the form `Tag-Sort(V1,...,VN)` where `Tag` represents the token identity of the structure using a Prolog variable, `Sort` is the type of structure, and `V1` through `VN` are the values of the appropriate features, in alphabetical order of the features' names, which are themselves left implicit. Furthermore, the `Tag` is used for forwarding and dereferencing during unification.

```
functional sub [forward,backward]
           intro [arg:synsem,
                  res:synsem].
  forward sub [].
  backward sub [].
```

This tells us that `functional` objects have `ARG` and `RES` features. Because `forward` and `backward` are subtypes of `functional`, they will also have `ARG` and `RES` features, with the same restrictions.

There are a couple of important restrictions placed on appropriateness conditions in ALE. The most significant of these is the acyclicity requirement. This condition disallows type specifications which require a type to have a value which is of the same or more specific type. For example, the following specification is *not* allowed:

```
person sub [male,female]
       intro [father:male,
              mother:female].
  male sub [].
  female sub [].
```

The problem here is the obvious one that there are no most general feature structures that are both of type `person` and totally well-typed.[5] This is because any person must have a father and mother feature, which are male and female respectively, but since male and female are subtypes of person, they must also have mother and father values. It is significant to note that the acyclicity condition does not rule out recursive structures, as can be seen with the example of lists. The `list` type specification is acceptable because not every list is required to have a head and tail, only non-empty lists are. The acyclicity restriction can be stated graph theoretically by constructing a directed graph from the type specification. The nodes of the graph are simply the types. There is an edge from every type to all of its supertypes, and an edge from every type to the types in the type restrictions in its features. Type specifications are only acceptable if they produce a graph with no cycles. One cycle in the `person` graph is from `male` to `person` (by the supertype relation) and from `person` to `male` (by the `FATHER` feature). On the other hand, there are no cycles in the specification of `list`.

The second restriction placed on appropriateness declarations is designed to limit non-determinism in much the same way as the bounded completeness condition on the inheritance hierarchy. This second condition requires every feature to be *introduced* at a unique most general type. In other words, the set of types appropriate for a feature must have a most general element. Thus the following type declaration fragment is *invalid*:

```
a sub [b,c,d].
  b sub []
    intro [f:w,
           g:x].
```

[5]The only finite feature structures that could meet this type system would have to be cyclic, as noted in Carpenter (1992). The problem is that there is no most general such cyclic structure, so type inference cannot be unique.

```
  c sub []
    intro [f:y,
           h:z].
  d sub [].
```

The problem is that the feature `F` is appropriate for types `b` and `c`, but there is not a unique most general type for which it's appropriate. In general, just like the bounded completeness condition, type specifications which violate the feature introduction condition can be patched, without violating any of their existing structure, by adding additional types. In this case, we add a new type between `a` and the types `b` and `c`, producing the equivalent well-formed specification:

```
a sub [e,d].
  e sub [b,c]
    intro [f:bot].
    b sub []
      intro [f:w,
             g:x].
    c sub []
      intro [f:y,
             h:z].
  d sub [].
```

This example also illustrates how subtypes of a type can place additional restrictions on values on features as well as introducing additional features.

As a further illustration of how feature introduction can be obeyed in general, consider the following specification of a type system for representing first-order terms:

```
sem_obj sub [individual,proposition].
  individual sub [a,b].
    a sub [].
    b sub [].
proposition sub [atomic_prop,relational].
  atomic_prop sub [].
  relational_prop sub [unary_prop,transitive_prop]
                  intro [arg1:individual].
    unary_prop sub [].
    transitive_prop sub [binary_prop,ternary_prop]
                    intro [arg2:individual].
      binary_prop sub [].
      ternary_prop sub []
                   intro [arg3:individual].
```

In this case, unary propositions have one argument feature, binary propositions have two argument features, and ternary propositions have three argument features, all of which must be filled by individuals.

3.6 Extensionality

ALE also respects the distinction between *intensional* and *extensional* types (see Carpenter (1992:Chapter 8). The concept of extensional typing has its origins in the assumption in standard treatments of feature structures, that there can only be one copy of any *atom* (a feature structure with no appropriate features) in a feature structure. Thus, if path π_1 leads to atom a, and path π_2 leads to atom a, then the values for those two paths are token-identical. Token-identity refers to an identity between two feature structures as objects, as opposed to *structure-identity*, which refers to an identity between two feature structures that contain the same information.

Smolka (1988a) partitioned his atoms according to whether more than one copy could exist or not. In ALE, following Carpenter (1992), this notion of copyability has been extended to arbitrary types – loosely speaking, those types which are copyable we call *intensional*, and those which are not we call *extensional*. Thus, it is possible to have two feature structures of the same intensional type which, although they may be structure-identical, are not token-identical. Formally:

> Given the set of types, Type, defined by an ALE signature, we designate a subset, ExtType $\subseteq$ Type, as the set of extensional types. With the exception of `a_/1` atoms, discussed below, this set consists only of *maximally specific types*, i.e., for each $\sigma \in$ ExtType, there is no type τ such that σ subsumes τ.

The restriction of ExtType to maximally specific types is peculiar to ALE, and is levied in order to reduce the computational complexity of enforcing extensionality.[6]

We need one more definition to formally state the effect which an extensional type has on feature structures in ALE.

> Given a set of extensional types, ExtType, we define an equivalence relation, $\asymp$, the *collasping relation*, on well-typed feature structures, such that $F_1 \asymp F_2$ for $F_1 \neq F_2$ only if:
>
> - F_1 has the same type, σ, as F_2, and $\sigma \in$ ExtType, and
> - for every feature, f, appropriate to σ, F_1^f, the value of f in F_1, and F_2^f, the value of f in F_2, are defined, and $F_1^f \asymp F_2^f$.

In ALE, all feature structures behave as if they are what Carpenter (1992) referred to as *collapsed*. That is, the only collapsing relation that exists between any two feature structures is the trivial collapsing relation, namely:

> $F_1 \asymp F_2$ if and only if F_1 is token-identical to F_2.

In the case of acyclic feature structures, this definition is equivalent to saying that two feature structures of the same extensional type are token-identical if and only if, for every feature appropriate to that type, their respective values on that feature are token-identical. For example, supposing that we have a signature representing

[6] In theory (Carpenter 1992), this set is only required to be *upward closed*, which means that if $\sigma \in$ ExtType, and σ subsumes τ, then $\tau \in$ ExtType. This relaxation of our requirement that extensional types be maximal would actually not be too difficult to implement.

dates, then the two substructures representing dates in the following structure must be token identical.

```
married_person
BIRTHDAY [1] date
         DAY  12
         MONTH nov
         YEAR 1971
SPOUSE BIRTHDAY [1] date
                DAY 12
                MONTH nov
                YEAR 1971
```

In other words, this represents a person born on 12 November 1971, who is married to a person with the same birthdate.

Now consider a slightly more complex example, which employs the following signature.

```
bot sub [a,b,c,g].
  a sub []
    intro [f:b,g:c].
  b sub [].
  c sub [].
  g sub []
    intro [h:a,j:a].
```

If `a`, `b`, and `c` are extensional, then the values of `H` and `J` in `g` are always token-identical, i.e., every feature structure of type `g` satisfies:

```
g
H [0] a
  F b
  G c
J [0]
```

But if only `a`, and `b` are extensional, and `c` is intensional, then the values of `H` and `J` are not necessarily token-identical, although they are always structure-identical:

```
g
H a
  F [1] b
  G c
J a
  F [1]
  G c
```

To cite an earlier example, suppose we were to specify that the type `false`, used in the liar sentence and its negation, were extensional. Now the liar sentence's representation is:

```
[0] false
    ARG1 [0]
```

as before, but the negation of the liar sentence would also be represented by:

```
[0] false
    ARG1 [0]
```

since if were still represented by:

```
[1] false
ARG1 [0] false
     ARG1 [0]
```

then we could cite a non-trivial collasping relation, $\asymp$, in which $[1] \asymp [0]$.

As a related example, consider:

```
s
A [0] t
  C [0]
B [1] t
  C [1]
```

Assuming that `t` is extensional and only appropriate for the feature `C`, then the structures `[0]` and `[1]` in the above structure would be identified.

Extensionality allows the proper representation of feature structures and terms in both PATR-II, the Rounds-Kasper system, and in Prolog and Prolog II. For PATR-II and the Rounds-Kasper system, all atoms (those types with no appropriate features) are assumed to be extensional. Furthermore, in the Rounds-Kasper and PATR-II systems, which are monotyped, there is only one type that is appropriate for any features, and it must be appropriate for all features in the grammar. In Prolog and Prolog II, the type hierarchy is assumed to be flat, and every type is extensional.

Just as with implementations of Prolog, collapsing is only performed as it is needed. As shown by Carpenter (1992), collapsing can be restricted to cases where inequations are tested between two structures, with exactly the same failure behavior. It turns out to be less efficient to collapse structures before asserting them into ALE's parsing chart, primarily because the time to test arbitrary structures for collapsibility is at least quadratic in the size of the structures being collapsed. See the section below on inequations for further discussion. Currently, extensionality is only enforced before the answer to a user query is given.

Extensional types in ALE are specified all at once in a list:

`ext`$([ext_1, \ldots, ext_n])$.

in the same file in which the subsumption relation is defined. All types that are not mentioned in the `ext` specification are assumed to be intensional, except ALE's `a_/1` atoms, discussed below, which have the same extensionality as Prolog terms, i.e., if they are ground or have the same variables in the same positions.[7] These do not need to be declared as such. If more than one `ext` specification is given, the first one is used. If no `ext` specification is given, then the specification:

```
ext([]).
```

[7] This is given by the `==` operator in Prolog.

is assumed. If a type occurs in `ext/1`, but does not appear on the left or right-hand side of a `sub` declaration, it assumed to be maximal, and immediately subsumed by `bot`.

Of course, collapsing is only enforced between feature structures whose life-spans in ALE[8] overlap. So, for example, if one request is made for the representation of the liar sentence:

```
[0] false
    ARG1 [0]
```

and then another is made for that of its negation, the output is not:

```
[0]
```

(referring to the same token above) but rather:

```
[0] false
    ARG1 [0]
```

Every time a new context arises, numbering of structures begins again from `[0]`.

3.7 `a_/1` Atoms

ALE also provides an infinite collection of atoms. These are of the form:

`a_` *Term*

where *Term* is a prolog term. Two `a_/1` atoms subsume each other if and only if their terms subsume each other as prolog terms. As a result, no two different, ground atoms subsume each other; and the most general atom of this collection is `a_ _`. They implicitly exist in every type hierarchy, with `a_ _` being immediately subsumed by `bot`, and with every ground atom being maximal. `a_/1` atoms are extensional; and non-ground `a_/1` atoms are extensionally identical as Prolog terms, i.e., if they have the same variables in the same positions. For example, `a_ f(X)` and `a_ g(X)` are not taken to be the same atom, nor are `a_ f(X)` and `a_ f(f(X)`, nor are `a_ f(X)` and `a_ f(Y)`. But `a_ f(X)` and `a_ f(X)` are. Their status in the type hierarchy should not be explicitly declared, nor should the fact that they bear no features, nor should their extensionality.

Some care must be exercised when using non-ground atoms in chart edges. ALE's chart parser copies edges, including the Prolog variables inside `a_/1` atoms. When these variables are copied, identity among variables within a single edge is preserved, but identity among variables between different edges may be lost. Because ALE delays the enforcement of extensional type checking, this could result in ALE losing a path equation between two atoms. The best way to avoid this is always to use ground atoms in chart edges. Otherwise, the user should at least avoid relying on extensional identity when writing grammars by not using the ALE built-in `@=` or inequations between non-ground atoms from different edges.

[8] The life-span of a feature structure in ALE is the period from its creation to the point when the user command currently being executed finishes, unless that feature structure is asserted as an edge in ALE's chart parser. In this case, the life of the feature structure ends when the edge is removed. Every new request for a parse to ALE removes all of the current edges.

If the user requires intensional atoms, they must be explicitly declared. There must also be no user-defined type, `a_`, in the type hierarchy. Certain arguments to `a_/1` cannot be used, such as `a_` itself, and other prolog reserved words, such as `mod`, unless they are used with the proper operator precedence and proper number of arguments to be parsed by Prolog.

Otherwise, `a_/1` atoms can be used wherever a normal type can. They are particularly useful as members of large domains that are too tedious to define, such as phonology attributes in natural language grammars, or to pass extra-logical information around a parse tree, such as numbers representing probabilities. To declare a feature's value as any `a_/1` atom, use `a_ _`:

```
sign intro [phon:(a_ _)].
```

The parentheses are recommended for readability, but not necessary. Because subsumption among `a_/1` atoms mirrors subsumption as prolog terms, one can also declare features appropriate for only certain kinds of atoms. For example:

```
sign intro [phon:(a_ phon(_))].
```

declares PHON appropriate to any atom whose term's functor is `phon/1`.

Structure-sharing between `a_/1` prolog terms in feature appropriateness declarations is ignored by ALE. For example, the declaration:

```
foo intro [f:(a_ X),g:(a_ X)].
```

is treated as:

```
foo intro [f:(a_ _),g:(a_ _)].
```

ALE does respect structure-sharing between `a_/1` prolog terms in descriptions.

3.8 Attribute-Value Logic

Now that we have seen how the type system must be specified, we turn our attention to the specification of feature structures themselves. The most convenient and expressive method of describing feature structures is the logical language developed by Kasper and Rounds (1986), which we modify here in three ways. First, we replace the notion of path sharing with the more compact and expressive notion of variable due to Smolka (1988a). Second, we extend the language to types, following Pollard (in press). Finally, we add inequations.

The collection of *descriptions* used in ALE can be described by the following BNF grammar:

```
<desc> ::= <type>
         | <variable>
         | (<feature>:<desc>)
         | (<desc>,<desc>)
         | (<desc>;<desc>)
         | (=\= <desc>)
```

As we have said before, both types and features are represented by Prolog constants. Variables, on the other hand, are represented by Prolog variables. As indicated by the BNF, no whitespace is needed around the feature selecting colon, conjunction comma and disjunction semi-colon, but any whitespace occurring will be ignored.

These descriptions are used for picking out feature structures that satisfy them. We consider the clauses of the definition in turn. A description consisting of a type picks out all feature structures of that type. A variable can be used to refer to any feature structure, but multiple occurrences of the same variable must refer to the same structure. A description of the form `(<feature>:<desc>)` picks out a feature structure whose value for the feature satisfies the nested description. An inequation `=\= <desc>` is satisfied by those feature structures that are not token-identical to the feature structure described by `<desc>`. Inequations are discussed in more detail below.

There are two ways of logically combining descriptions: following Prolog, the comma represents conjunction and the semi-colon represents disjunction. A feature structure satisfies a conjunction of descriptions just in case it satisfies both conjuncts, while it satisfies a disjunction of descriptions if it satisfies either of the disjuncts.

We should also add to the above BNF grammar the following line:

```
<desc> ::= (<path> == <path>)
```

This is an equational description, of which inequations are the negation. Equational or inequational descriptions are satisfied by the presence or absence, respectively, of token-identity. In particular, an inequation between two structurally-identical feature structures can be satisfied, while a path equation can *only* be satisfied by two structurally-identical feature structures, but is not necessarily satisfied.

All instances of equational descriptions can be captured by using multiple occurrences of variables. For example, the description:

```
([arg1]==[arg2])
```

is equivalent to the description:

```
(arg1:X,arg2:X).
```

assuming there are no other occurrences of `X`.

Standard assumptions about operator precedence and association are followed by ALE, allowing us to omit most of the parentheses in descriptions. In particular, equational descriptions bind the most tightly, followed by feature selecting colon, then by inequations, then conjunction and finally disjunction. Furthermore, conjunction and disjunction are left-associative, while the feature selector is right-associative. For instance, this gives us the following equivalences between descriptions:

```
a, b ; c, d ; e = (a,b);(c,d);e

a,b,c = a,(b,c)

f:g:bot,h:j = (f:(g:bot)),(h:j)

f:g: =\=k,h:j = (f:(g: =\=(k))),(h:j)

f:[g]==[h],h:j = (f:([g]==[h])),(h:j)
```

Note that a space must occur between `=\=` and other operators such as `:`.

A description may be satisfied by no structure, a finite number of structures or an infinite collection of feature structures. A description is said to be *satisfiable* if it is satisfied by at least one structure. A description ϕ *entails* a description ψ if every structure satisfying ϕ also satisfies ψ. Two descriptions are *logically equivalent* if they entail each other, or equivalently, if they are satisfied by exactly the same set of structures.

ALE is only sensitive to the differences between logically equivalent formulas in terms of speed. For instance, the two descriptions `(tl:list,ne_list,hd:bot)` and `hd:bot` are satisfied by exactly the same set of totally well-typed structures assuming the type declaration for lists given above, but the smaller description will be processed much more efficiently. There are also efficiency effects stemming from the order in which conjuncts (and disjuncts) are presented. The general rule for speedy processing is to eliminate descriptions from a conjunction if they are entailed by other conjuncts, and to put conjuncts with more type and feature entailments first. Thus with our specification for relations above, the description `(arg1:a, binary_proposition)` would be slower than `(binary_proposition,arg1:a)`, since `binary_proposition` entails the existence of the feature `arg1`, but not conversely.[9]

At run-time, ALE computes a representation of the most general feature structure that satisfies a description. Thus a description such as `hd:a` with respect to the list grammar is satisfied by the structure:

```
ne_list
HD a
TL list
```

Every other structure satisfying the description `hd:a` is subsumed by the structure given above. In fact, the above structure is said to be a *vague* representation of all of the structures that satisfy the description. The type conditions in ALE were devised to obey the very important property, first noted by Kasper and Rounds (1986), that every non-disjunctive description is satisfied by a unique most general feature structure. Thus in the case of `hd:a`, there is no more general feature structure than the one above which also satisfies `hd:a`.

The previous example also illustrates the kind of *type inference* used by ALE. Even though the description `hd:a` does not explicitly mention either the feature `TL` or the type `ne_list`, to find a feature structure satisfying the description, ALE must infer this information. In particular, because `ne_list` is the most general type for which `HD` is appropriate, we know that the result must be of type `ne_list`. Furthermore, because `ne_list` is appropriate for both the features `HD` and `TL`, ALE must add an appropriate `TL` value. The value type `list` is also inferred, due to the fact that a `ne_list` must have a `TL` value which is a list. As far as type inference goes, the user does not need to provide anything other than the type specification; the system computes type inference based on the appropriateness specification. In general, type inference is very efficient in terms of time. The biggest concern should

[9] This is because the depth of dereferencing depends on the history of types a given structure is instantiated to. There is no path compression on-line, but it is carried out before an edge is asserted into the chart.

be how large the structures become.[10] In contrast to a vague description, a disjunctive description is usually *ambiguous*. Disjunction is where the complexity arises in satisfying descriptions, as it corresponds operationally to non-determinism.[11] For instance, the description `hd:(a;b)` is satisfied by two distinct minimal structures, neither of which subsumes the other:

```
ne_list       ne_list
HD a          HD b
TL list       TL list
```

On the other hand, the description `hd:atom` is satisfied by the structure:

```
ne_list
HD atom
TL list
```

Even though the descriptions `hd:atom` and `hd:(a;b)` are not logically equivalent (though the former entails the latter), they have the interesting property of being unifiable with exactly the same set of structures. In other words, if a feature structure can be unified with the most general satisfier of `hd:atom`, then it can be unified with one of the minimal satisfiers of `hd:(a;b)`.

In terms of efficiency, it is very important to use vagueness wherever possible rather than ambiguity. In fact, it is almost always a good idea to arrange the type specification with just this goal in mind. For instance, consider the difference between the following pair of type specifications, which might be used for English gender:

```
gender sub [masc,fem,neut].
  masc sub [].
  fem sub [].
  neut sub [].
```

```
gender sub [animate,neut].
  animate sub [masc,fem].
    masc sub [].
    fem sub [].
  neut sub [].
```

Now consider the fact that the relative pronouns *who* and *which* are distinguished on the basis of whether they select animate or inanimate genders. In the flatter hierarchy, the only way to select the animate genders is by the ambiguous description `masc;fem`. The hierarchy with an explicit animate type can capture the same possibilities with the vague description `animate`. An effective rule of thumb is that ALE does an amount of work at best proportional to the number of most general satisfiers of a description and at worst proportional to 2^n, where n is the number of disjuncts in the description. Thus the ambiguous description requires roughly twice the time and memory to process than the vague description. Whether the amount of work is closer to the number of satisfiers or exponential in the number of disjuncts depends on how many unsatisfiable disjunctive possibilities drop out early in the computation.

[10] Finding most general satisfiers for non-disjunctive descriptions, even those involving type inference, is quasi-linear in the size of the description. But it should be kept in mind that there is also a factor of complexity determined by the size of the type specification. In practice, this factor is proportional to how large the inferred structure is. In general, the size of the inferred structure is linear in the size of the description, with a constant for the type specification.

[11] It corresponds so closely with non-determinism that satisfiability of descriptions with disjunctions is NP-complete. Furthermore, the algorithm employed by ALE may produce up to 2^n satisfiers for a description with n disjunctions.

3.8.1 Enforcement of Inequations

Inequations are persistent in that once that are created, they remain as long as one of the structures being inequated remains. Thus the following two descriptions are logically equivalent:

```
f:(=\=c), f:c

f:c, f:(=\=c)
```

Both will cause failure; but they are not operationally equivalent. An inequation is evaluated when it arises, and again after high-level unifications in the system; but inequations are not evaluated every time an inequated structure is modified. In an ideal system, inequations would be attached directly to structures so that they could be evaluated on-line during unification. As things stand, ALE represents a feature structure as a regular feature structure with a separate set of inequations. Also, the complexity is sensitive to the conjunctive normal form of inequations at the time at which it is evaluated, though this form may later be reduced.

These sets of inequations are evaluated at run-time at the point they are encountered, before answers are given to top-level queries, before any edge is added to ALE's parsing chart, after every daughter is matched to a description in an ALE grammar rule[12], and after the head of an ALE definite clause has been matched to a goal. At compile-time, inequations are checked for every empty category, for every lexical entry, and after every lexical rule application.

Inequations are also symmetric. Thus the following two descriptions are logically equivalent:

```
f:(=\= X),g:X

f:X,g:(=\= X)
```

Both inequate the values of F and G. Again, these are not operationally equivalent. Because inequations are evaluated at the time they are encountered, the second ordering will normally detect an immediate failure *sooner* than the first.

An inequation between two feature structures is a requirement for them not to be *token-identical*. Thus, if a type is intensional, it is possible for two feature structures to be of that same type, and still satisfy an inequation between them. Thus, any attempt to inequate two structures that should be identical as a result of extensional typing will also cause failure. For instance, consider the following:

```
s                s
F [1] t          F [3]
  H [1]    +     G [4]            =  failure
G [2] t          [3] =\= [4]
  H [1]
```

The values of the features `F` and `G` cannot be inequated because they are extensionally identical (assuming the type `t` is declared to be extensional and is only appropriate for the feature `H`.

[12] In the case of `cats>`, they are enforced after the list description itself is matched, and also after every element of the list is matched.

When inequations are evaluated, they are reduced. This reduction consists, in part, of partial evaluation of extensionality checking, which is otherwise delayed in ALE. For instance, consider the following:

```
F [1] s
  H bot
  J bot
G [2] s
  H bot
  J bot
[1] =\= [2]
```

If the type `s` is extensional and appropriate for the features `H` and `J`, then the inequation above is reduced to the following:

```
F [1] s
  H [3] bot
  J [4] bot
G [2] s
  H [5]
  J [6]
[3] =\= [5] ; [4] =\= [6]
```

The set of inequations is stored in conjunctive normal form. The cost is some space over the re-evaluation of inequations. Of course, if the types on `[3]` and `[4]` were more refined than `bot`, then the inequations `[3] =\= [5]` and `[4] =\= [6]` would further reduce. In addition, when reducing inequations in this way, we eliminate those that are trivially satisfied. The ones that are printed are only the residue after reduction. For instance, consider the following structure:

```
F [1] s
  H [3] a
  J bot
G [2] s
  H [3]
  J bot
[1] =\= [2]
```

Since the `H` values are token-identical, this structure reduces to the following.

```
F s
  H [3] a
  J [4] bot
G s
  H [3]
  J [5] bot
[4] =\= [5]
```

If structures `[4]` and `[5]` had been of non-unifiable types, then there would be no residual inequation at all — the original inequation would trivially be satisfied.

An important subcase is that of an inequation between extensional atoms. If an atom is extensional, then there is only one instance of it. Thus an inequation between two identical, extensional atoms always fails. For example, if a type signature includes:

```
bot sub [..., a, b, ...].
a sub [].
  intro [f:bot].
b sub [].
...
ext([..., b, ...]).
```

then the description:

```
(a,f:=\= b)
```

is satisfied just in those cases where the value of F is not of type **b**. If **b** were intensional, then the inequation in this description would essentially have no effect. In fact, the only productive instances of inequations between two intensionally typed feature structures are those used with multiply occurring variables. In all other instances, there is no way for the inequation to be violated, since there is no way to render a structurally-identical copy of an intensionally typed feature structure token-identical to any other structure. ALE detects these trivially satisfied inequations and disposes of them.

3.9 Macros

ALE allows the user to employ a general form of parametric macros in descriptions. Macros allow the user to define a description once and then use a shorthand for it in other descriptions. We first consider a simple example of a macro definition, drawn from the categorial grammar in the appendix. Suppose the user wants to employ a description `qstore:e_list` frequently within a program. The following macro definition can be used in the program file:

```
quantifier_free macro
  qstore:e_list.
```

Then, rather than including the description `qstore:e_list` in another description, `@ quantifier_free` can be used instead. Whenever `@ quantifier_free` is used, `qstore:e_list` is substituted.

In the above case, the `<macro_spec>` was a simple atom, but in general, it can be supplied with arguments. The full BNF for macro definitions is as follows:

```
<macro_def> ::= <macro_head> macro <desc>.

<macro_head> ::= <macro_name>
               | <macro_name>(<seq(<var>)>)

<macro_spec> ::= <macro_name>
               | <macro_name>(<seq(<desc>)>)
```

```
<seq(X)> ::= X
           | X, <seq(X)>
```

Note that `<seq(X)>` is a parametric category in the BNF which abbreviates non-empty sequences of objects of category `X`. The following clause should be added to recursive definition of descriptions:

```
<desc> ::=  @ <macro\_spec>
```

A feature structure satisfies a description of the form `@ <macrospec>` just in case the structure satisfies the body of the definition of the macro.

Again considering the categorial grammar in the appendix, we have the following macros with one and two arguments respectively:

```
np(Ind) macro
  syn:np,
  sem:Ind.

n(Restr,Ind) macro
  syn:n,
  sem:(body:Restr,
       ind:Ind).
```

In general, the arguments in the definition of a macro must be Prolog variables, which can then be used as variables in the body of the macro. With the first macro, the description `@ np(j)` would then be equivalent to the description `syn:np,sem:j`. When evaluating a macro, the argument supplied, in this case `j`, is substituted for the variable when expanding the macro. In general, the argument to a macro can itself be an arbitrary description (possibly containing macros). For instance, the description:

```
n((and,conj1:R1,conj2:R2),Ind3)
```

would be equivalent to the description:

```
syn:n,
sem:(body:(and,conj1:R1,conj2:R2),
     ind:Ind3)
```

This example illustrates how other variables and even complex descriptions can be substituted for the arguments of macros. Also note the parentheses around the arguments to the first argument of the macro. Without the parentheses, as in `n(and,conj1:R1,conj2:R2,Ind3)`, the macro expansion routine would take this to be a four argument macro, rather than a two argument macro with a complex first argument. This brings up a related point, which is that different macros can have the same name as long as they have the different numbers of arguments.

Macros can also contain other macros, as illustrated by the macro for proper names in the categorial grammar:

```
pn(Name) macro
  synsem: @ np(Name),
  @ quantifier_free.
```

In this case, the macros are expanded recursively, so that the description `pn(j)` would be equivalent to the description

```
synsem:(syn:np,sem:j),qstore:e_list
```

It is usually a good idea to use macros whenever the same description is going to be re-used frequently. Not only does this make the grammars and programs more readable, it reduces the number of simple typing errors that lead to inconsistencies.

As is to be expected, macros can't be recursive. That is, a macro, when expanded, is not allowed to invoke itself, as in the *ill-formed* example:

```
infinite_list(Elt) macro
  hd:Elt,
  tl:infinite_list(Elt)
```

The reason is simple; it is not possible to expand this macro to a finite description. Thus all recursion must occur in grammars or programs; it can't occur in either the appropriateness conditions or in macros.

The user should note that variables in the scope of a macro are not the same as ALE feature structure variables — they denote where macro-substitutions of parameters are made, *not* instances of re-entrancy in a feature structure. If we employ the following macro:

```
blah(X) macro
  b,
  f: X,
  g: X.
```

with the argument `(c,h:a)` for example we obtain the following feature structure:

```
b
F c
  H a
G c
  H a
```

where the values of `F` and `G` are not shared (unless `c` and `a` are extensional). We can, of course, create a shared structure using `blah`, by including an ALE variable in the actual argument to the macro. Thus `blah((Y,c,h:a))` yields:

```
b
F [0] c
  H a
G [0]
```

Because programming with lists is so common, ALE has a special macro for it, based on the Prolog list notation. A description may also take any of the forms on the left, which will be treated equivalently to the descriptions on the right in the following diagram:

```
[]                  e_list

[H|T]               (hd:H,
                     tl:T)

[A1,A2,...,AN]      (hd:A1,
                     tl:(hd:A2,
                         tl: ...
                                 tl:(hd:AN,
                                     tl:e_list)...))

[A1,...,AN|T]        (hd:A1,
                     tl:(hd:A2,
                         tl: ...
                                 tl:(hd:AN,
                                     tl:T)...))
```

Note that this built-in macro does not require the macro operator `@`. Thus, for example, the description `[a|T3]` is equivalent to `hd:a,tl:T3`, and the description `[a,b,c]` is equivalent to `hd:a,tl:(hd:b,tl:(hd:c,tl:e_list))`. There are many example of this use of Prolog's list notation in the grammars in the appendix.

3.10 Functional Descriptions

ALE also provides the means to define functions mapping descriptions to descriptions. The syntax is:

```
<func_def> ::= <func_spec> +++> <desc>.

<func_spec> ::= <func_name>
              | <func_name>(<seq(desc)>)
```

Functional descriptions are compiled into code that is evaluated at run-time, first adding to each argument (if any) the description given for that argument, and then evaluating to the resulting description, which can itself include other functional descriptions, including recursive calls. As an example, one may consider the `append/2` function:

```
append([],L) +++> L.
append([X|L1],L2) +++> [X|append(L1,L2)].
```

The lists shown here are instances of ALE's list macro notation. Notice that the only type checking in this definition is performed by appropriateness and the list macro, so the first clause could succeed with any `L`. To add more, one could redefine the first clause as:

```
append([],(list,L)) +++> L.
```

Functional descriptions can be used wherever other descriptions can. The user should be particularly careful with ensuring that the arguments to a functional

description call will be sufficiently instantiated for the call to terminate; and the clauses, correctly ordered for the description to terminate correctly. Type checking cannot ensure this, in general. For example, in the definition for `append/2`, with or without explicit type-checking on the second argument, both clauses will match a functional description whose first argument or any TL value in the first argument is strictly of type `list`, i.e., a list that is not known to be `elist` or `nelist`. Such a functional description will, thus, evaluate to an infinite number of results. Clauses in functional descriptions are considered in the order they are given; and the search for solutions *always* backtracks into subsequent clauses.

Any functional description that can be defined so that its argument descriptions are only variables, and its result has no (mutual) recursion should be defined as a macro instead, which is completely expanded at compile-time. Remember, however, that it is not always sufficient to replace the `+++>` with `macro`: variable replacement in macros works by true textual replacement, whereas the variables in functional descriptions are ALE descriptions. For example, the functional description:

```
foo(X) +++> (bar,f:X,g:X).
```

evaluates to a feature structure with a re-entrancy. The macro:

```
foo(X) macro (bar,f:X,g:X).
```

does not necessarily evaluate to a feature structure with a re-entrancy. The functional description, `foo/1` is correctly converted to the macro:

```
foo(X) macro (bar,f:(Y,X),g:(Y,X)).
```

The ALE variable, `Y`, establishes the re-entrancy that the macro variable, `X`, does not.

3.11 Type Constraints

Our logical language of descriptions can be used with the type system in order to enforce constraints on feature structures of a particular type. Constraints are attached to types, and may consist of arbitrary descriptions. Their effect is to require every structure of the constrained type to always satisfy the constraint description.

Constraints are enforced using the `cons` operator, e.g.:

```
bot sub [a,b].
  a sub []
    intro [f:b,g:b].
  b sub [].
a cons (f:X,g:=\= X).
```

The constraint on the type `a` (which must occur within parentheses) requires all feature structures of type `a` to have non-token-identical values for features f and g. Notice that the type `b` has no constraints expressed. This is equivalent to specifying the constraint:

```
b cons bot.
```

which is satisfied by any feature structure (of type `b`). A type constraint may use any of the operators in the description language, including further type descriptions, which may themselves be constrained. The type, `bot`, may not have type constraints, nor may `a_/1` atoms.

It is crucial that the type descriptions be finitely resolvable. Because simple depth-first search is used to evaluate constraints, infinite resolution paths will cause the system to hang. For example, the following signature should not contain the following constraints:

```
bot sub [a,b].
  a sub [c]
    intro [f:bot].
    c sub [].
  b sub []
    intro [g:bot].
a cons f:b.
b cons g:c.
```

This is because `a` subsumes `c`. Notice, however, that type constraints can be used to provide additional information regarding value restrictions on appropriate features. In general, ALE performs more efficiently when restrictions are provided in the appropriateness conditions, rather than in general constraints; but type constraints can encode a greater variety of restrictions. Specifically, they allow constraints to express path equations and inequations, as well as deeper path restrictions. Constraints may include relational constraints, which are defined using definite clauses, which are discussed below. Type constraints are efficiently compiled in the same way as other descriptions. Also, like appropriateness conditions, they are only enforced once for any given structure.

It is also important to note that because of the delay in ALE's inequational enforcement, type constraints that involve recursion that terminates by an inequation failure may go into infinite loops due to this delay in enforcement. Because extensionality is only enforced before the answer to a top-level query is given, recursive type constraints that rely on the extensional identity of two feature structures to terminate on the basis of their type will not terminate.

3.12 Example: The Zebra Puzzle

We now provide a simplified form of the Zebra Puzzle (Figure 3.1), a common puzzle for constraint resolution. This puzzle was solved by Aït-Kaci (1984) using roughly the same methods as we use here. The puzzle illustrates the expressive power of the combination of extensional types, inequations and type constraints. Such puzzles, sometimes known as logic puzles or constraint puzzles, require one to find a state of affairs within some situation that simultaneously satisfies a set of constraints. The situation itself very often implicitly levies certain constraints.

We can represent the simplified Zebra Puzzle in ALE as:

```
% Subsumption
%======================
```

Puzzle: Three different houses each contain a different pet, a different drink, and an inhabitant of a different nationality. The following six facts are true about these houses:

1. The man in the third (middle) house drinks milk.
2. The Spaniard owns the dog.
3. The Ukranian drinks the tea.
4. The Norwegian lives in the first house.
5. The Norwegian lives next to the tea-drinker.
6. The juice-drinker owns the fox.

Questions: Who owns the zebra? Who drinks juice?

Figure 3.1: The Zebra Puzzle.

```
bot sub [house,descriptor,background].

  descriptor sub [nat_type,ani_type,bev_type].
    nat_type sub [norwegian,ukranian,spaniard].
      norwegian sub [].
      ukranian sub [].
      spaniard sub [].
    ani_type sub [fox,dog,zebra].
      fox sub [].
      dog sub [].
      zebra sub [].
    bev_type sub [juice,tea,milk].
      juice sub [].
      tea sub [].
      milk sub [].

  house sub []
      intro [nationality:nat_type,animal:ani_type,beverage:bev_type].

  background sub [clue]
             intro [house1:house,house2:house,house3:house].
    clue sub [maximality].
      maximality sub [].

ext([norwegian,ukranian,spaniard,fox,dog,zebra,juice,tea,milk]).

% Constraints
%============================
background cons
  (house1:nationality:N1,                          % inequational constraints
```

```
    house2:nationality:(N2,(=\= N1)),
    house3:nationality:((=\= N1),(=\= N2)),

    house1:animal:A1,
    house2:animal:(A2,(=\= A1)),
    house3:animal:((=\= A1),(=\= A2)),

    house1:beverage:B1,
    house2:beverage:(B2,(=\= B1)),
    house3:beverage:((=\= B1),(=\= B2))).

  clue cons
    (house3:beverage:milk,                                % clue 1

     (house1:nationality:spaniard,house1:animal:dog       % clue 2
     ;house2:nationality:spaniard,house2:animal:dog
     ;house3:nationality:spaniard,house3:animal:dog),

     (house1:nationality:ukranian,house1:beverage:tea     % clue 3
     ;house2:nationality:ukranian,house2:beverage:tea
     ;house3:nationality:ukranian,house3:beverage:tea),

     house1:nationality:norwegian,                        % clue 4

     (house1:nationality:norwegian,house2:beverage:tea    % clue 5
     ;house2:nationality:norwegian,house3:beverage:tea
     ;house2:nationality:norwegian,house1:beverage:tea
     ;house3:nationality:norwegian,house2:beverage:tea),

     (house1:beverage:juice,house1:animal:fox             % clue 6
     ;house2:beverage:juice,house2:animal:fox
     ;house3:beverage:juice,house3:animal:fox)).


  maximality cons
    (house1:nationality:(norwegian;ukranian;spaniard), % maximality constraints
     house2:nationality:(norwegian;ukranian;spaniard),
     house3:nationality:(norwegian;ukranian;spaniard),

     house1:animal:(fox;dog;zebra),
     house2:animal:(fox;dog;zebra),
     house3:animal:(fox;dog;zebra),

     house1:beverage:(juice;tea;milk),
     house2:beverage:(juice;tea;milk),
     house3:beverage:(juice;tea;milk)).
```

The subsumption hierarchy is shown pictorially in Figure 3.2. The type,

`background`, with the assistance of the types subsumed by `house` and `descriptor`, represents the situation of three houses (the features of `background`), each of which has three attributes (the features of `house`). The implicit constraints levied by the situation appear as constraints on the type, `background`, namely that no two houses can have the same value for any attribute. These are represented by inequations. But notice that, since we are interested in treating the values of attributes as tokens, we must represent those values by extensional types. If we had not done this, then we could still, for example, have two different houses with the beverage, `juice`, since there could be two different feature structures of type `juice` that were not token-identical. Notice also that all of these types are maximal, and hence satisfy the restriction that ALE places on extensional types.

The explicit constraints provided by the clues to the puzzle are represented as constraints on the type `clue`, a subtype of `background`. We also need a subtype of `clue`, `maximality`, to enforce another constraint implicit to all constraint puzzles, namely the one which requires that we provide only maximally specific answers, rather than vague solutions which say, for example, that the beverage for the third house is a type of beverage (`bev_type`), which may actually still satisfy a puzzle's constraints.

To solve the puzzle, we simply type:

```
| ?- mgsat maximality.

MOST GENERAL SATISFIER OF: maximality

maximality
HOUSE1 house
       ANIMAL fox
       BEVERAGE juice
       NATIONALITY norwegian
HOUSE2 house
       ANIMAL zebra
       BEVERAGE tea
       NATIONALITY ukranian
HOUSE3 house
       ANIMAL dog
       BEVERAGE milk
       NATIONALITY spaniard


ANOTHER?  y.

no
| ?-
```

So the Ukranian owns the zebra, and the Norwegian drinks juice. A most general satisfier of `maximality` will also satisfy the constraints of its supertypes, `background` and `clue`.

Although ALE is capable of representing such puzzles and solving them, it is not actually very good at solving them efficiently. To solve such puzzles efficiently, a

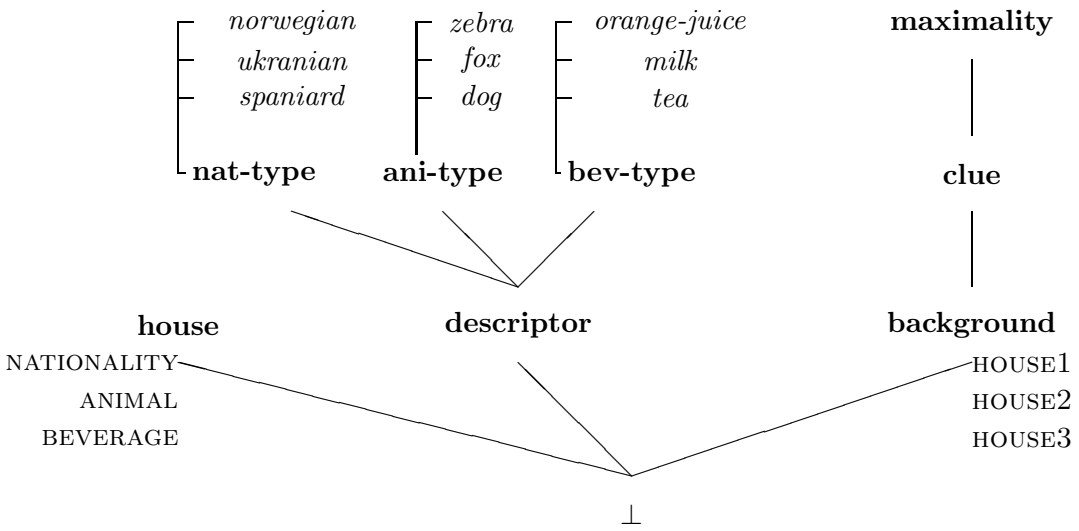

Figure 3.2: Inheritance Network for the Zebra Puzzle.

system must determine an optimal order in which to satisfy all of the various constraints. ALE does not do this since it can express definite clauses in its constraints, and the reordering would also be very difficult for the user to keep track of while designing a grammar. A system that does do this is the general constraint resolver proposed by Penn and Carpenter (1993)[13].

[13]This system was actually the precursor to ALE. It implemented a completely reversible constraint-based parser/generator with a weighting on the constraints based on their maximal non-determinism. Re-ordering constraints, however, proved to be insufficient for efficient parsing or generation, compared to a uni-directional system such as ALE.

Chapter 4

Definite Clauses

The next two sections, covering the constraint logic programming and phrase structure components of ALE, simply describe how to write ALE programs and how they will be executed. Discussion of interacting with the system itself follows the description of the programming language ALE provides.

The definite logic programming language built into ALE is a constraint logic programming (CLP) language, where the constraint system is the attribute-value logic described above. Thus, it is very closely related to both Prolog and LOGIN. Like Prolog, definite clauses may be defined with disjunction, negation and cut. The definite clauses of ALE are executed in a depth-first, left to right search, according to the order of clauses in the database. ALE performs last call optimization, but does not perform any clause indexing.[1] Those familiar with Prolog should have no trouble adapting that knowledge to programming with definite clauses in ALE. The only significant difference is that first-order terms are replaced with descriptions of feature structures.

While it is not within the scope of this user's guide to detail the logic programming paradigm, much less CLP, this section will explain all that the user familiar with logic programming needs to know to exploit the special features of ALE. For background, the user is encouraged to consult Sterling and Shapiro (1986) with regard to general logic programming techniques, most of which are applicable in the context of ALE, and Aït-Kaci and Nasr (1986) for more details on programming with sorted feature structures. For more advanced material on programming in Prolog with a compiler, see O'Keefe (1990). The general theory of CLP is developed in a way compatible with ALE in Höhfeld and Smolka (1988). Of course, since ALE is literally an implementation of the theory found in Carpenter (1992), the user is strongly encouraged to consult Chapter 14 of that book for full theoretical details.

The syntax of ALE's logic programming component is broadly similar to that of Prolog, with the only difference being that first-order terms are replaced with attribute-value logic descriptions. The language in which clauses are expressed in ALE is given in BNF as:

```
<clause> ::= <literal> if <goal>.

<literal> ::=  <pred_sym>
```

[1] Thus, additional cuts might be necessary to ensure determinism, so that last call optimization is effective.

```
              |  <pred_sym>(<seq(desc)>)

<goal> ::= true
         | <literal>
         | (<goal>,<goal>)
         | (<goal>;<goal>)
         | (<desc> =@ <desc>)
         | (<cut_free_goal> -> <goal>)
         | (<cut_free_goal> -> <goal> ; <goal>)
         | !
         | (\+ <goal>)
         | prolog(<prolog_goal>)
```

Just as in Prolog, predicate symbols must be Prolog atoms. This is a more restricted situation than the definite clause language discussed in Carpenter (1992), where literals were also represented as feature structures and described using attribute-value logic. Also note that ALE requires every clause to have a body, which might simply be the goal `true`. There must be whitespace around the `if` operator, but none is required around the conjunction comma, the disjunction semicolon, the cut or shallow cut symbols `!,->`, or the unprovability symbol `\+`. Parentheses, in general, may be dropped and reconstructed based on operator precedences. The precedence is such that the comma binds more tightly than the semicolon, while the unprovability symbol binds the most tightly. Both the semicolon and comma are right associative.

The operational behavior of ALE is nearly identical to Prolog with respect to goal resolution. That is, it evaluates a sequence of goals depth-first, from the left to right, using the order of clauses established in the program. The only difference arises from the fact that, in Prolog, terms cannot introduce non-determinism. In ALE, due to the fact that disjunctions can be nested inside of descriptions, additional choice points might be created both in matching literals against the heads of clauses and in expanding the literals within the body of a clause. In evaluating these choices, ALE maintains a depth-first left to right strategy.

We begin with a simple example, the `member/2` predicate:[2]

```
member(X,hd:X) if
  true.
member(X,tl:Xs) if
  member(X,Xs).
```

As in Prolog, ALE clauses may be read logically, as implications, from right to left. Thus the first clause above states that `X` is a member of a list if it is the head of a list. The second clause states that `X` is a member of a list if `X` is a member of the tail of the list, `Xs`. Note that variables in ALE clauses are used the same way as in Prolog, due to the notational convention of our description language. Further note that, unlike Prolog, ALE requires a body for every clause. In particular, note that the first clause above has the trivial body `true`. The compiler is clever enough to remove such goals at compile time, so they do not incur any run-time overhead.

[2] As in Prolog, we refer to predicates by their name and arity.

Given the notational convention for lists built into ALE, the above program could equivalently be written as:

```
member(X,[X|_]) if
  true.
member(X,[_|Xs]) if
  member(X,Xs).
```

But recall that ALE would expand `[X|_]` as `(hd:X,tl:_)`. Not only does ALE not support anonymous variable optimizations, it also creates a conjunction of two descriptions, where `hd:X` would have sufficed. Thus the first method is not only more elegant, but also more efficient.

Due to the fact that lists have little hierarchical structure, list manipulation predicates in ALE look very much like their correlates in Prolog. They will also execute with similar performance. But when the terms in the arguments of literals have more interesting taxonomic structure, ALE actually provides a gain over Prolog's evaluation method, as pointed out by Aït-Kaci and Nasr (1986). Consider the following fragment drawn from the syllabification grammar in the appendix, in which there is a significant interaction between the inheritance hierarchy and the definite clause `less_sonorous/2`:

```
segment sub [consonant,vowel].
  consonant sub [nasal,liquid,glide].
    nasal sub [n,m].
      n sub [].
      m sub [].
    liquid sub [l,r].
      l sub [].
      r sub [].
    glide sub [y,w].
      y sub [].
      w sub [].
  vowel sub [a,e,i]
    a sub [].
    e sub [].
    i sub [].

less_sonorous_basic(nasal,liquid) if true.
less_sonorous_basic(liquid,glide) if true.
less_sonorous_basic(glide,vowel) if true.

less_sonorous(L1,L2) if
  less_sonorous_basic(L1,L2).
less_sonorous(L1,L2) if
  less_sonorous_basic(L1,L3),
  less_sonorous(L3,L2).
```

For instance, the third clause of `less_sonorous_basic/2`, being expressed as a relation between the types `glide` and `vowel`, allows solutions such as

`less_sonorous_basic(w,e)`, where `glide` and `vowel` have been instantiated as the particular subtypes `w` and `e`. This fact would not be either as straightforward or as efficient to code in Prolog, where relations between the individual letters would need to be defined. The loss in efficiency stems from the fact that Prolog must either code all 14 pairs represented by the above three clauses and type hierarchy, or perform additional logical inferences to infer that `w` is a glide, and hence less sonorous than the vowel `e`. ALE, on the other hand, performs these operations by unification, which, for types, is a simple table look-up.[3] All in all, the three clauses for `less_sonorous_basic/2` given above represent relations between 14 pairs of letters. Of course, the savings is even greater when considering the transitive closure of `less_sonorous_basic/2`, given above as `less_sonorous/2`, and would be greater still for a type hierarchy involving a greater degree of either depth or branching.

While we do not provide examples here, suffice it to say that cuts, shallow cuts, negation, conjunction and disjunction work exactly the same as they do in Prolog. In particular, cuts conserve stack space representing backtracking points, disjunctions create choice points and negation is evaluated by failure, with the same results on binding as in Prolog.

The definite clause language also allows arbitrary prolog goals, using the predicate, `prolog(<prolog_goal>)`. This is perhaps most useful when used with the Prolog predicates, `assert` and `retract`, which provide ALE users with access to the Prolog database, and with I/O statements, which can be quite useful for debugging definite clauses.

Should `prolog_goal` contain a variable that has been instantiated to an ALE feature structure, this will appear to Prolog as ALE's internal representation of that feature structure. Feature structures can be asserted and retracted, however, without regard to their internal structure. The details of ALE's internal representation of feature structures can be found in Carpenter and Penn (1996), and briefly on p. 85.

Another side effect of not directly attaching inequations to feature structures is that if a feature structure with inequations is asserted and a copy of it is later instantiated from the Prolog database or retracted, the copy will have lost the inequations. In general, passing feature structures with inequations to Prolog hooks should be avoided.

Because the enforcement of extensionality is delayed in ALE, a variable which is instantiated to an extensionally typed feature structure and then passed to a prolog hook may also not reflect token identities as a result of extensionality. Provided that there are no inequations (to which the user does not have direct access), this can be enforced within the hook by calling the ALE internal predicate `extensionalise(FS,[])`.

There is a special literal predicate, `=@`, used with infix notation, which is true when its arguments are token-identical. As with inequations, which forbid token-identity, the `=@` operator is of little use unless multiply occurring variables are used in its arguments' descriptions. Note, however, that while inequations (`=\=`) and path equations (`==`) are part of the description language, `=@` is a definite clause predicate, and cannot be used as a description. It is more important to note that while the negation of the structure-identity operator (`==`), namely the inequation

[3] Table look-ups involved in unification in ALE rely on double hashing, once for the type of each structure being unified.

(=\=), is monotonic when interpreted persistently, the negation of the token-identity operator (=@), achieved by using it inside the argument of the \+ operator, is non-monotonic, and thus its use should be avoided.

It is significant to note that clauses in ALE are truly definite in the sense that only a single literal is allowed as the *head* of a clause, while the *body* can be a general goal. In particular, disjunctions in descriptions of the arguments to the head literal of a clause are interpreted as taking wide scope over the entire clause, thus providing the effect of multiple solutions rather than single disjunctive solutions. The most simple example of this behavior can be found in the following program:

```
foo((b;c)) if true.

bar(b) if true.

baz(X) if foo(X), bar(X).
```

Here the query `foo(X)` will provide two distinct solutions, one where `X` is of type `b`, and another where it is of type `c`. Also note that the queries `foo(b)` and `foo(c)` will succeed. Thus the disjunction is equivalent to the two single clauses:

```
foo(b) if true.
foo(c) if true.
```

In particular, note that the query `baz(X)` can be solved, with `X` instantiated to an object of type `b`. In general, using embedded disjunctions will usually be more efficient than using multiple clauses in ALE, especially if the disjunctions are deeply nested late in the description. On the other hand, cuts can be inserted for control with multiple clauses, making them more efficient in some cases.

4.1 Type Constraints Revisited

The type constraints mentioned in the last chapter can also incorporate relational constraints defined by definite clauses, with the optional operator `goal`. Consider the following example from HPSG:

```
word cons W
     goal (single_rel_constraint(W),
           clausal_rel_prohibition(W)).
```

In this example, the constraint from the description language is simply the variable `W`, which is used to match any feature structure of type `word`. That feature structure is then passed as an argument to the two procedural attachments `single_rel_constraint/1` and `clausal_rel_prohibition/1`, which each represent a principle from HPSG which governs words (among other objects). Notice that the goal, when non-literal, must occur within parentheses.

While every type constraint must have a description, procedural attachments are optional. If they do occur, they occur after the description. The syntax is given in BNF as:

```
<cons_spec> ::= <type> cons <desc>
              | <type> cons <desc>
                       goal <goal>
```

Chapter 5

Phrase Structure Grammars

The ALE phrase structure processing component is loosely based on a combination of the functionality of the PATR-II system and the DCG system built into Prolog. Roughly speaking, ALE provides a system like that of DCGs, with two primary differences. The first difference stems from the fact that ALE uses attribute-value logic descriptions of typed feature structures for representing categories and their parts, while DCGs use first-order terms (or possibly cyclic variants thereof). The second primary difference is that ALE's parser uses a bottom-up active chart parsing algorithm and a semantic-head-driven generator rather than encoding grammars directly as Prolog clauses and evaluating them top-down and depth-first. In the spirit of DCGs, ALE allows definite clause procedures to be attached and evaluated at arbitrary points in a phrase structure rule, the difference being that these rules are given by definite clauses in ALE's logic programming system, rather than directly in Prolog.

Phrase structure grammars come with two basic components, one for describing lexical entries and empty categories, and one for describing grammar rules. We consider these components in turn.

5.1 Lexical Entries

Lexical entries in ALE are specified as rewriting rules, as given by the following BNF syntax:

```
<lex_entry> ::=  <word> ---> <desc>.
```

For instance, in the categorial grammar lexicon in the appendix, the following lexical entry is provided, along with the relevant macros:

```
john --->
  @ pn(j).

pn(Name) macro
  synsem: @ np(Name),
  @ quantifier_free.

np(Ind) macro
```

```
  syn:np,
  sem:Ind.

quantifier_free macro
  qstore:[].
```

Read declaratively, this rule says that the word `john` has as its lexical category the most general satisfier of the description `@ pn(j)`, which is:

```
cat
SYNSEM basic
       SYN np
       SEM j
QSTORE e_list
```

Note that this lexical entry is equivalent to that given without macros by:

```
john --->
  synsem:(syn:np,
          sem:j),
  qstore:e_list.
```

Macros are useful as a method of organizing lexical information to keep it consistent across lexical entries. The lexical entry for the word `runs` is:

```
runs ---> @ iv((run,runner:Ind),Ind).

iv(Sem,Arg) macro
  synsem:(backward,
          arg: @ np(Arg),
          res:(syn:s,
               sem:Sem)),
  @ quantifier_free.
```

This entry uses nested macros along with structure sharing, and expands to the category:

```
cat
SYNSEM backward
       ARG synsem
           SYN np
           SEM [0] sem_obj
       RES SYN s
           SEM run
               RUNNER [0]
QSTORE e_list
```

It also illustrates how macro parameters are in fact treated as variables.

Multiple lexical entries may be provided for each word. Disjunctions may also be used in lexical entries, but are expanded out at compile-time. Thus the first three lexical entries, taken together, compile to the same result as the fourth:

```
bank --->
  syn:noun,
  sem:river_bank.
bank --->
  syn:noun,
  sem:money_bank.
bank --->
  syn:verb,
  sem:roll_plane.

bank --->
  ( syn:noun,
    sem:( river_bank
        ; money_bank
        )
  ; syn:verb,
    sem:roll_plane
  ).
```

Note that this last entry uses the standard Prolog layout conventions of placing each conjunct and disjunct on its own line, with commas at the end of lines, and disjunctions set off with vertically aligned parentheses at the beginning of lines.

The compiler finds all the most general satisfiers for lexical entries at compile time, reporting on those lexical entries that have unsatisfiable descriptions. In the above case of `bank`, the second combined method is marginally faster at compile-time, but their run-time performance is identical. The reason for this is that both entries have the same set of most general satisfiers.

ALE supports the construction of large lexica, as it relies on Prolog's hashing mechanism to actually look up a lexical entry for a word during bottom-up parsing. For generation, ALE indexes lexical entries for faster unification, as described in Penn and Popescu (1997). Constraints on types can also be used to enforce conditions on lexical representations, allowing for further factorization of information.

5.2 Empty Categories

ALE allows the user to specify certain categories as occurring without any corresponding surface string. These are usually referred to somewhat misleadingly as *empty categories*, or sometimes as *null productions*. In ALE, they are supported by a special declaration of the form:

```
empty <desc>.
```

Where `<desc>` is a description of the empty category.

For example, a common treatment of bare plurals is to hypothesize an empty determiner. For instance, consider the contrast between the sentences *kids overturned my trash cans* and *a kid overturned my trash cans*. In the former sentence, which has a plural subject, there is no corresponding determiner. In our categorial grammar, we might assume an empty determiner with the following lexical entry (presented here with the macros expanded):

```
empty @ gdet(some).

gdet(Quant) macro
  synsem:(forward,
          arg:(syn:(n,
                    num:plu),
               sem:(body:Restr,
                    ind:Ind)),
          res:(syn:(np,
                    num:plu),
               sem:Ind),
  qstore:[ (Quant,
            var:Ind,
            restr:Restr) ].
```

Of course, it should be noted that this entry does not match the type system of the categorial grammar in the appendix, as it assumes a number feature on nouns and noun phrases.

Empty categories are expensive to compute under a bottom-up parsing scheme such as the one used in ALE. The reason for this is that these categories can be used at every position in the chart during parsing (with the same begin and end points). If the empty categories cause local structural ambiguities, parsing will be slowed down accordingly as these structures are calculated and then propagated. Consider the empty determiner given above. It can be used as an inactive edge at every node in the chart, then match the forward application rule scheme and search through every edge to its right looking for a nominal complement. If there are relatively few nouns in a sentence, not many noun phrases will be created by this rule and thus not many structural ambiguities will propagate. But in a sentence such as *the kids like the toys*, there will be an edge spanning *kids like the toys* corresponding to an empty determiner analysis of *kids*. The corresponding noun phrase created spanning *toys* will not propagate any further, as there is no way to combine a noun phrase with the determiner *the*. But now consider the empty slash categories of form X/X in GPSG. These categories, when coupled with the slash passing rules, would roughly double parsing time, even for sentences that can be analyzed without any such categories. The reason is that these empty categories are highly underspecified and thus have many options for combinations. Thus empty categories should be used sparingly, and prefarably in environments where their effects will not propagate.

Another word of caution is in order concerning empty categories: they can occur in constructions with other empty categories. For instance, if we specify categories C_1 and C_2 as empty categories, and have a rule that allows a C to be constructed from a C_1 and a C_2, then C will act as an empty category, as well. These combinations of empty categories are computed at compile-time; but the sheer number of empty categories produced under this closure may be a processing burden if they apply at run-time too productively. Keep in mind that ALE computes all of the inactive edges that can be produced from a given input string, so there is no way of eliminating the extra work produced by empty categories interacting with other categories, including empty ones.

5.3 Lexical Rules

Lexical rules provide a mechanism for expressing redundancies in the lexicon, such as the kinds of inflectional morphology used for word classes, derivational morphology as found with suffixes and prefixes, as well as zero-derivations as found with detransitivization, nominalization of some varieties and so on. The format ALE provides for stating lexical rules is similar to that found in both PATR-II and HPSG.

In order to implement them efficiently, lexical rules, as well as their effects on lexical entries, are compiled in much the same way as grammars. To enhance their power, lexical rules, like grammar rules, allow arbitrary procedural attachment with ALE definite constraints.

The lexical rule system of ALE is productive in that it allows lexical rules to apply sequentially to their own output or the output of other lexical rules. Thus, it is possible to derive the nominal *runner* from the verb *run*, and then derive the plural nominal *runners* from *runner*, and so on. At the same time, the lexical system is leashed to a fixed depth-bound, which may be specified by the user. This bound limits the number of rules that can be applied to any given category. The bound on application of rules is specified by a command such as the following, which should appear somewhere at the beginning of the input file:

```
:-lex_rule_depth(2).
```

Of course, bounds other than `2` can be used. The bound indicates how many applications of lexical rules can be made, and may be `0`. If there is more than one such specification in an input file, the last one will be the one that is used. If no specification is given, the default is `2`.

The format for lexical rules is as follows:

```
<lex_rule> ::= <lex_rule_name> lex_rule <lex_rewrite>
                               morphs <morphs>.

<lex_rewrite> ::= <desc> **> <desc>
                | <desc> **> <desc> if <goal>

<morphs> ::= <morph>
           | <morph>, <morphs>

<morph> ::= (<string_pattern>) becomes (<string_pattern>)
          | (<string_pattern>) becomes (<string_pattern>)
                               when <prolog_goal>

<string_pattern> ::= <atomic_string_pattern>
                   | <atomic_string_pattern>, <string_pattern>

<atomic_string_pattern> ::=  <atom>
                           |  <var>
                           |  <list(<var_char>)>
<var_char> ::= <char>
             | <var>
```

An example of a lexical rule with almost all of the bells and whistles (we put off procedural attachment for now) is:

```
plural_n lex_rule
  (n,
   num:sing)
   **> (n,
        num:plu)
  morphs
    goose becomes geese,
    [k,e,y] becomes [k,e,y,s],
    (X,man) becomes (X,men),
    (X,F) becomes (X,F,es) when fricative(F),
    (X,ey) becomes (X,[i,e,s]),
    X becomes (X,s).

  fricative([s]).
  fricative([c,h]).
  fricative([s,h]).
  fricative([x]).
```

We will use this lexical rule to explain the behavior of the lexical rule system. First note that the name of a lexical rule, in this case `plural_n`, must in general be a Prolog atom. Further note that the top-level parentheses around both the descriptions and the patterns are necessary. If the Prolog goal, in this case `fricative(F)`, had been a complex goal, then it would need to be parenthesized as well. The next thing to note about the lexical rule is that there are two descriptions — the first describes the input category to the rule, while the second describes the output category. These are arbitrary descriptions, and may contain disjunctions, macros, etc. We will come back to the clauses for `fricative/1` shortly. Note that the patterns in the morphological component are built out of variables, sequences and lists. Thus a simple rewriting can be specified either using atoms as with `goose` above, with a list, as in `[k,e,y]`, or with a sequence as in `(X,man)`, or with both, as in `(X,[i,e,s])`. The syntax of the morphological operations is such that in sequences, atoms may be used as a shorthand for lists of characters. But lists must consist of variables or single characters only. Thus we could not have used `(X,[F])` in the fricative case, as `F` might is itself a complex list such as `[s,h]` or `[x]`. But in general, variables ranging over single characters can show up in lists.

The basic operation of a lexical rule is quite simple. First, every lexical entry, including a word and a category, that is produced during compilation is checked to see if its category satisfies the input description of a lexical rule. If it does, then a new category is generated to satisfy the output description of the lexical rule, if possible. Note that there might be multiple solutions, and all solutions are considered and generated. Thus multiple solutions to the input or output descriptions lead to multiple lexical entries.

After the input and output categories have been computed, the word of the input lexical entry is fed through the morphological analyzer to produce the corresponding output word. Unlike the categorial component of lexical rules, only one output word

will be constructed, based on the first input/output pattern that is matched.[1] The input word is matched against the patterns on the left hand side of the morphological productions. When one is found that the input word matches, any condition imposed by a `when` clause on the production is evaluated. This ordering is imposed so that the Prolog goal will have all of the variables for the input string instantiated. At this point, Prolog is invoked to evaluate the `when` clause. In the most restricted case, as illustrated in the above lexical rule, Prolog is only used to provide abbreviations for classes. Thus the definition for `fricative/1` consists only of unit clauses. For those unfamiliar with Prolog, this strategy can be used in general for simple morphological abbreviations. Evaluating these goals requires the `F` in the input pattern to match one of the strings given. The shorthand of using atoms for the lists of their characters only operates within the morphological sequences. In particular, the Prolog goals do not automatically inherit the ability of the lexical system to use atoms as an abbreviation for lists, so they have to be given in lists. Substituting `fricative(sh)` for `fricative([s,h])` would not yield the intended interpretation. Variables in sequences in morphological productions will always be instantiated to lists, even if they are single characters. For instance, consider the lexical rule above with every atom written out as an explicit list:

```
[g,o,o,s,e] becomes [g,e,e,s,e],
[k,e,y] becomes [k,e,y,s],
(X,[m,a,n]) becomes (X,[m,e,n]),
(X,F) becomes (X,F,[e,s]) when fricative(F),
(X,[e,y]) becomes (X,[i,e,s]),
X becomes (X,[s]).
```

In this example, the `s` in the final production is given as a list, even though it is only a single character.

The morphological productions are considered one at a time until one is matched. This ordering allows a form of suppletion, whereby special forms such as those for the irregular plural of `goose` and `key` to be listed explicitly. It also allows subregularities, such as the rule for fricatives above, to override more general rules. Thus the input word `beach` becomes `beaches` because `beach` matches `(X,F)` with `X = [b,e,a]` and `F = [c,h]`, the goal `fricative([c,h])` succeeds and the word `beaches` matches the output pattern `(X,F,[e,s])`, instantiated after the input is matched to `([b,e,a],[c,h],[e,s])`. Similarly, words that end in `[e,y]` have this sequence replaced by `[i,e,s]` in the plural, which is why an irregular form is required for `keys`, which would otherwise match this pattern. Finally, the last rule matches any input, because it is just a variable, and the output it produces simply suffixes an `[s]` to the input.

For lexical rules with no morphological effect, the production:

```
X becomes X
```

suffices. To allow lexical operations to be stated wholly within Prolog, a rule may be used such as the following:

[1] Thus ALE's lexical rule system is not capable of handling cases of partial suppletion, where both a regular and irregular morphological form are both allowed. To allow two ouptut forms, one must be coded by hand with its own lexical entry or a separate lexical rule.

```
  X becomes Y when morph_plural(X,Y)
```

In this case, when `morph_plural(X,Y)` is called, `X` will be instantiated to the list of the characters in the input, and as a result of the call, `Y` should be instantiated to a ground list of output characters.

We finally turn to the case of lexical rules with procedural attachments, as in the following (simplified) example from HPSG:

```
extraction lex_rule
  local:(cat:(head:H,
              subcat:Xs),
         cont:C),
  nonlocal:(to_bind:Bs,
            inherited:Is)
  **> local:(cat:(head:H,
                  subcat:Xs2),
             cont:C),
      nonlocal:(to_bind:Bs,
                inherited:[G|Is])
  if
    select(G,Xs,Xs2)
  morphs
    X becomes X.

  select(X,(hd:X),Xs) if true.
  select(X,[Y|Xs],[Y|Ys]) if
    select(X,Xs,Ys).
```

This example illustrates an important point other than the use of conditions on categories in lexical rules. The point is that even though only the `LOCAL CAT SUBCAT` and `NONLOCAL INHERITED` paths are affected, information that stays the same must also be mentioned. For instance, if the `cont:C` specification had been left out of either the input our output category description, then the output category of the rule would have a completely unconstrained content value. This differs from the default-based nature of the usual presentation of lexical rules, which assumes all information that hasn't been explicitly specified is shared between the input and the output. As another example, we must also specify that the `HEAD` and `TO_BIND` features are to be copied from the input to the output; otherwise there would be no specification of them in the output of the rule. This fact follows from the description of the application of lexical rules: they match a given category against the input description and produce the most general category(s) matching the output description.

Turning to the use of conditions in the above rule, the `select/3` predicate is defined so that it selects its first argument as a list member of its second argument, returning the third argument as the second argument with the selected element deleted. In effect, the above lexical rule produces a new lexical entry which is like the original entry, except for the fact that one of the elements on the subcat list of the input is removed from the subcat list and added to the inherited value in the output. Nothing else changes.

Procedurally, the definite clause is invoked after the lexical rule has matched the input description against the input category. Like the morphological system, this

control decision was made to ensure that the relevant variables are instantiated at the time the condition is resolved. The condition here can be an arbitrary goal, but if it is complex, there should be parentheses around the whole thing. Cuts should not be used in conditions on lexical rules (see the comments on cuts in grammar rules below, which also apply to cuts in lexical rules).

Currently, ALE does not check for redundancies or for entries that subsume each other, either in the base lexicon or after closure under lexical rules. ALE also does not apply lexical rules to empty categories.

5.4 Grammar Rules

Grammar rules in ALE are of the phrase structure variety, with annotations for both goals that need to be solved and for attribute-value descriptions of categories. The BNF syntax for rules is as follows:

```
<rule> ::=  <rule_name> rule <desc> ===> <rule_body>.

<rule_body> ::=  <rule_clause>
               |  <rule_clause>, <rule_body>

<rule_clause> ::=  cat> <desc>
                 |  cats> <desc>
                 |  sem_head> <desc>
                 |  goal> <goal>
                 |  sem_goal> <goal>
```

The <rule_name> must be a Prolog atom. The description in the rule is taken to be the mother category in the rule, while the rule body specifies the daughters in the rule along with any side conditions on the rule, expressed as ALE goals. A further restriction on rules, which is not expressed in the BNF syntax above, is that there must be at least one category-seeking rule clause in each rule body.[2] Thus empty productions are not allowed and will be flagged as errors at compile time.

A simple example of such a rule, without any goals, is as follows:

```
s_np_vp rule
(syn:s,
 sem:(VPSem,
      agent:NPSem))
===>
cat>
  (syn:np,
   agr:Agr,
   sem:NPSem),
cat>
  (syn:vp,
   agr:Agr,
   sem:VPSem).
```

[2]By doubling the size of the BNF for rules, this requirement could be expressed.

There are a few things to notice about this rule. The first is that the parentheses around the category and mother descriptions are necessary. Looking at what the rule means, it allows the combination of an `np` category with a `vp` type category if they have compatible (unifiable) values for `agr`. It then takes the semantics of the result to be the semantics of the verb phrase, with the additional information that the noun phrase semantics fills the agent role.

Unlike the PATR-II rules, but similar to DCG rules, "unifications" are specified by variable co-occurrence rather than by path equations, while path values are specified using the colon rather than by a second kind of path equation. The rule above is similar to a PATR-II rule which would look roughly as follows:

```
x0 ---> x1, x2 if
  (x0 syn) == s,
  (x1 syn) == np,
  (x2 syn) == vp,
  (x0 sem) == (x2 sem),
  (x0 sem agent) == (x1 sem),
  (x1 agr) == (x2 agr)
```

Unlike lexical entries, rules are not expanded to feature structures at compile-time. Rather, they are compiled down into structure-copying operations involving table look-ups for feature and type symbols, unification operations for variables, sequencing for conjunction, and choice point creation for disjunction.

The descriptions for `cat>` and `cats>` daughters are always evaluated in the order they are specified, from left to right. This is significant when considering goals that might be interleaved with searches in the chart for consistent daughter categories. The order in which the code for the mother's and semantic head's descriptions is executed depends on the control strategy used during parsing or generation. These are described in Sections 5.4.3 and 5.4.4, respectively. In theory, the same grammar can be used for both parsing and generation. In practice, a single grammar is rarely efficient in both directions, and can even exhibit termination problems in one, just as a Prolog program may have these problems with queries that have different argument instantiations. So while it is not necessary to fully understand the parsing or generation algorithms used by ALE to exploit its power for developing grammars, practical implementations will order their procedural attachments and distribute their description-level information with these algorithms in mind.

Within a single description, in the case of feature and type symbols, a double-hashing is performed on the type of the structure being added to, as well as either the feature or the type being added. Additional operations arise from type coercions that adding features or types require. Thus there is nothing like disjunctive normal-form conversion of rules at compile time, as there is for lexical entries. In particular, if there is a local disjunction in a rule, it will be evaluated locally at run time. For instance, consider the following rule, which is the local part of HPSG's Schema 1:

```
schema1 rule
(cat:(head:Head,
      subcat:[]),
 cont:Cont)
===>
```

```
cat>
 (Subj,
  cat:head:( subst
           ; spec:HeadLoc,
           )),
cat>
 (HeadLoc,
  cat:(head:Head,
       subcat:[Subj]),
  cont:Cont).
```

Note that there is a disjunction in the `cat:head` value of the first daughter category (the subject in this case). This disjunction represents the fact that the head value is either a substantive category (one of type `subst`), or it has a specifier value which is shared with the entire second daughter. But the choice between the disjuncts in the first daughter of this rule is made locally, when the daughter category is fully known, and thus does not create needless rule instantiations.

ALE's general treatment of disjunction in descriptions, which is an extension of Kasper and Round's (1986) attribute-value logic to phrase structure rules, is a vast improvement over a system such as PATR-II, which would not allow disjunction in a rule, thus forcing the user to write out complete variants of rules that only differ locally. Disjunctions in rules do create local choice points, though, even if the first goal in the disjunction is the one that is solvable.[3] This is because, in general, both parts of a disjunction might be consistent with a given category, and lead to two solutions. Or one disjunct might be discarded as inconsistent only when its variables are further instantiated elsewhere in the rule.

5.4.1 Procedural Attachments

A more complicated rule, drawn from the categorial grammar in the appendix is as follows:

```
backward_application rule
(synsem:Z,
 qstore:Qs)
===>
cat>
  (synsem:Y,
   qstore:Qs1),
cat>
  (synsem:(backward,
           arg:Y,
           res:Z),
   qstore:Qs2),
goal>
  append(Qs1,Qs2,Qs).
```

[3] In a future release, cuts will be allowed within descriptions, to allow the user to eliminate disjunctive choice points when possible.

Note that the goal in this rule is specified after the two category descriptions. Consequently, it will be evaluated after categories matching the descriptions have already been found, thus ensuring in this case that the variables `Qs1` and `Qs2` are instantiated. The `append(Qs1,Qs2,Qs)` goal is evaluated by ALE's definite clause resolution mechanism. `goal>` attachments are always evaluated in the order they are specified relative to the enforcement of `cat>` and `cats>` daughters, from left to right. All possible solutions to the goal are found with the resulting instantiations carrying over to the rule. These solutions are found using the depth-first search built into ALE's definite constraint resolver. In general, goals may be interleaved with the category specifications, giving the user control over when the goals are fired. Also note that goals may be arbitrary *cut-free* ALE definite clause goals, and thus may include disjunctions, conjunctions, and negations. Cuts may occur, however, within the code for any literal clause specified in a procedural attachment. The attachments themselves must be cut-free to avoid the cut taking precedence over the entire rule after compilation, thus preventing the rule to apply to other edges in the chart or for later rules to apply. Instead, if cuts are desired in rules, they must be encapsulated in an auxiliary predicate, which will restrict the scope of the cut. For instance, in the context of a phrase structure rule, rather than a goal of the form:

```
goal>
  (a, !, b)
```

it is necessary to encode this as follows:

```
goal>
  c
```

where the predicate `c` is defined by:

```
c if
  (a, !, b).
```

This prevents backtracking through the cut in the goal, but does not block the further application of the rule. A similar strategy should be employed for cuts in lexical rules.

As a programming strategy, rules should be formulated like Prolog clauses, so that they fail as early as possible. Thus the features that discriminate whether a rule is applicable should occur first in category descriptions. The only work incurred by testing whether a rule is applicable is up to the point where it fails.

Just as with PATR-II, ALE is RE-complete (equivalently, Turing-equivalent), meaning that any computable language can be encoded. Thus it is possible to represent undecidable grammars, even without resorting to the kind of procedural attachment possible with arbitrary definite clause goals. With its mix of depth-first and breadth-first evaluation strategies, ALE is not strictly complete with respect to its intended semantics if an infinite number of edges can be generated with the grammar. This situation is similar to that in Prolog, where a declaratively impeccable program might hang operationally.

5.4.2 The cats> Operator

The cats> operator is used to describe a list of daughters, whose length cannot be determined until run-time. Daughters are not parsed or generated as quickly as part of a cats> specification. Note also the interpretation of cats> requires that its argument is subsumed by the type list, which must be defined, along with ne_list, e_list, etc., and the features HD, and TL, which we defined above. This check is not made using unification, so that an underspecified list argument will not work either. If the argument of cats> is not subsumed by list, then the rule in which that argument occurs will never match any string, and a run-time error message will be given. This operator is useful for so-called "flat" rules, such as HPSG's Schema 2, part of which is given (in simplified form) below:

```
schema2 rule
(cat:(head:Head,
      subcat:[Subj]))
===>
cat>
 (cat:(head:Head,
       subcat:[Subj|Comps])),
cats> Comps.
```

Since various lexical items have SUBCAT lists of various lengths, e.g., zero for proper nouns, one for intransitive verbs, two for transitive verbs, cats> is required in order to match the actual list of complements at run-time.

It is common to require a goal to produce an output for the argument of cats>. If this is done, the goal must be placed before the cats>. Our use of cats> is problematic in that we require the argument of cats> to evaluate to a list of fixed length. Thus parsing with the following head-final version of HPSG's Schema 2 would not work:

```
schema2 rule
(cat:(head:Head,
      subcat:[SubjSyn]))
===>
cats> Comps,
cat>
 (cat:(head:Head,
       subcat:[Subj|Comps])).
```

One way to work around this is to place some finite upper bound on the size of the Comps list by means of a constraint.

```
schema2 rule
(cat:(head:Head,
      subcat:[SubjSyn]))
goal>  three_or_less(Comps),
cats> Comps,
cat>
 (cat:(head:Head,
```

```
        subcat:[Subj|Comps])).

three_or_less([]) if true.
three_or_less([_]) if true.
three_or_less([_,_]) if true.
three_or_less([_,_,_]) if true.
```

The problem with this strategy from an efficiency standpoint is that arbitrary sequences of three categories will be checked at every point in the grammar; in the English case, the search is directed by the types instantiated in `Comps` as well as that list's length. From a theoretical standpoint, it is impossible to get truly unbounded length arguments in this way.

5.4.3 Parsing

The ALE system employs a bottom-up active chart parser that has been tailored to the implementation of attribute-value grammars in Prolog. The single most important fact to keep in mind is that rules are evaluated from left to right, with the mother description coming last. Most of the implementational considerations follow from this rule evaluation principle and its specific implementation in Prolog. In parsing, `sem_head>` and `sem_goal>` specifications are treated exactly as `cat>` and `goal>` specifications, respectively.

The chart is filled in using a combination of depth- and breadth-first control. In particular, the edges are filled in from right to left, even though the rules are evaluated from left to right. Furthermore, the parser proceeds breadth-first in the sense that it incrementally moves through the string from right to left, one word at a time, recording all of the inactive edges that can be created beginning from the current left-hand position in the string. For instance, in the string `The kid ran yesterday`, the order of processing is as follows. First, lexical entries for `yesterday` are looked up, and entered into the chart as inactive edges. For each inactive edge that is added to the chart, the rules are also fired according to the bottom-up rule of chart parsing. But no active edges are recorded. Active edges are purely dynamic structures, existing only locally to exploit Prolog's copying and backtracking schemes. The benefit of parsing from right to left is that when an active edge is proposed by a bottom-up rule, every inactive edge it might need to be completed has already been found. This is actually true as long as the direction of traversal through the string is the opposite of the direction of matching daughter categories in a rule; thus the real reason for the right-to-left parsing strategy is to allow the active edges to be represented dynamically, while still evaluating the rules from left to right. While the overall strategy is bottom-up, and breadth-first insofar as it steps incrementally through the string, filling in every possible inactive edge as it goes, the rest of the processing is done depth-first to keep as many data structures dynamic as possible, to avoid copying other than that done by Prolog's backtracking mechanism. In particular, lexical entries, bottom-up rules, and active edges are all evaluated depth-first, which is perfectly sound, because they all start at the same left point (that before the current word in the right to left pass through the string), and thus do not interact with one another.

ALE computes the closure of its grammar rules under application of the first daughter's description to empty categories at compile-time. This is known as *Empty-*

First-Daughter closure or EFD closure. This closure operation has three advantages. First, given ALE's combination of depth-first and breadth-first processing, it is necessary in order to ensure completeness of parsing with empty categories, because any permutation of empty categories can, in principle, be combined to form a new empty category. Second, it works around a problem that many non-ISO-compatible Prologs, including SICStus Prolog, have with asserted predicates that results in empty category leftmost daughters not being able to combine with their own outputs. Third, it allows the run-time parser to establish a precondition that rules only need to be closed with non-empty leftmost daughters at run-time. As a result, when a new mother category is created and closed under rules as the leftmost daughter, it cannot combine with other edges created with the same left node. This allows ALE, at each step in its right-to-left pass throught the input string, to copy all of the edges in the internal database back onto the heap before they can be used again, and thus reduces edge copying to a constant two times per edge for non-empty categories. Keeping a copy of the chart on the heap also allows for more sophisticated indexing strategies that would otherwise be overwhelmed by the cost of copying edges with large-sized categories in Prolog before the match. The EFD closure algorithm itself is described in Penn (1999).

EFD closure potentially creates new rules, a prefix of whose daughters have matched empty categories, and new empty categories, formed when every daughter of a rule has matched an empty category. The closure in computed breadth-first.

EFD closure may not terminate. As a result, compilation of some grammars may go into infinite loops. This only occurs, however, with grammars for which every parse would go into an infinite loop at run-time if EFD closure were not applied — specifically, when empty categories alone can produce infinitely many empty categories using the rules of the grammar. Because early versions of ALE did not compute a complete closure of grammar rules over empty categories (even at run-time), some grammars that terminated at run-time under these early versions will not terminate at compile-time under the current version.

Rules can incorporate definite clause goals before, between or after category specifications. These goals are evaluated when they are found. For instance, if a goal occurs between two categories on the right hand side of a rule, the goal is evaluated after the first category is found, but before the second one is. The goals are evaluated by ALE's definite clause resolution mechanism, which operates in a depth-first manner. Thus care should be taken to make sure the required variables in a goal are instantiated before the goal is called. The resolution of all goals should terminate with a finite (possibly empty) number of solutions, taking into account the variables that are instantiated when they are called.

The parser will terminate after finding all of the inactive edges derivable from the lexical entries and the grammar rules. Of course, if the grammar is such that an infinite number of derivations can be produced, ALE will not terminate. Such an infinite number of derivations can creep in either through recursive unary rules or through the evaluation of goals.

ALE now has an optional mechanism for checking edge subsumption (Section 7.9). This can be used to prevent the propagation of spurious ambiguities through the parse. A category C spanning a given subsequence is said to be *spurious* if there is another category C' spanning the same subsequence such that C is subsumed by C'. Only the most general category needs to be recorded to ensure soundness. It

can also be used to detect two derivations of the same category. Our experience, however, has been that most unification-based grammars do not have any spurious ambiguity. They normally incorporate some notion of thematic or functional structure representing the meaning of a sentence; and in these cases, most structural ambiguities result in semantic ambiguities. For such grammars, subsumption checking is probably not worth the effort, and should be left disabled.

5.4.4 Generation

ALE also contains a generator, based on the *Semantic Head-Driven Generation* algorithm of van Noord (1989), as extended by Shieber et al. (1990), and adapted to the typed feature logic of Carpenter (1992) by Popescu (1996). Its implementation in ALE is described in Penn and Popescu (1997).

Given a description of a feature structure, ALE's generator will nondeterministically generate all the word strings that correspond to its most general satisfier(s). In other words, the generated word strings, when parsed in ALE using the same grammar, will result in feature structures that *unify* with a most general satisfier of the initial description (rather than necessarily be identical). That part of the feature structure which represents semantic information drives the generation process.

The `semantics/1` Directive

ALE identifies this part using a user-defined directive, `semantics/1`. This directive distinguishes a binary user-defined definite clause predicate as the predicate to use to find semantic information. The first argument is always the feature structure whose semantics are being identified; and the second argument is always the semantic information. The example below, taken again from the sample generation grammar, simply says that the semantics of a feature structure is the value of its `sem` feature:

```
semantics sem1.
sem1(sem:S,S) if true.
```

In general, the second argument does not need to be a sub-structure of the first — it could have a special type that is used only for the purpose of collecting semantic information, possibly spread over several unrelated sub-structures. The body can be arbitrarily complex; and there can be multiple clauses for the definition of this predicate. The predicate must, however, have the property that it will terminate when only its first argument is instantiated, and when only its second argument is instantiated. ALE will use this predicate in both "directions" — to find semantics information, and in reverse to build templates to find structures that have matching semantic information. There can be only one predicate distinguished by `semantics/1`. If there are multiple directives, ALE will only use the first.

The Algorithm

Semantic-head-driven generation makes use of the notion of a *semantic head* of a rule, a daughter whose semantics is shared with the mother. In semantic-head-driven generation, there are two kinds of rules: *chain rules*, which have a semantic

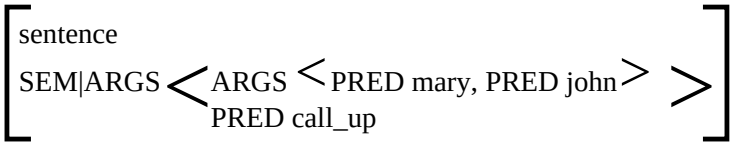

Figure 5.1: The initial root.

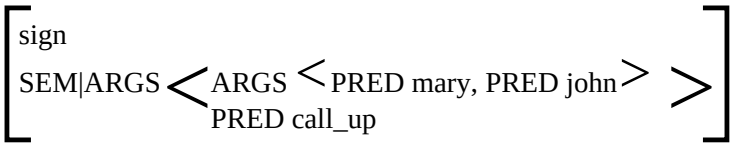

Figure 5.2: The semantics template.

head, and *non-chain rules*, which lack such a head. These two subsets are processed differently during the generation process.

Given a feature structure, called the *root* goal, to generate a string for, the generator builds a new feature structure that shares its semantic information (using the user-defined semantics predicate with the second argument instantiated) and finds a *pivot* that unifies with it. The pivot is the lowest node in a derivation tree that has the same semantics as the root. The pivot may be either a *lexical entry* or *empty category* (the base cases), or the *mother category of a non-chain rule*. Once a pivot is identified, one can recursively generate top-down from the pivot using non-chain rules. Since the pivot must be the *lowest*, there can be no lower semantic heads, and thus no lower chain-rule applications. Just as in parsing, the daughters of non-chain rules are processed from left to right.

Once top-down generation from the pivot is complete, the pivot is linked to the root bottom-up by chain rules. At each step, the current chain node (beginning with the pivot) is unified with the semantic head of a chain-rule, its non-head sisters are generated recursively, and the mother becomes the new chain node. The non-head daughters of chain rules are also processed from left to right. The base case is where the current chain node unifies with the root.

An example from the sample generation grammar in Appendix A.3 illustrates this better. Suppose that the generator is given the goal description:

```
(sentence,
 sem:(pred:decl,
      args:[(pred:call_up,
             args:[pred:mary,pred:john])]))
```

Figure 5.1 shows the initial root (the most general satisfier of the input description); and Figure 5.2 shows the template that ALE uses to find a pivot. Next (Figure 5.3), a pivot is selected, in this case by unifying the template with the mother category of the non-chain rule, `sentence1`:

```
sentence1 rule
  (sentence,sem:(pred:decl,
                 args:[S]))
  ===>
```

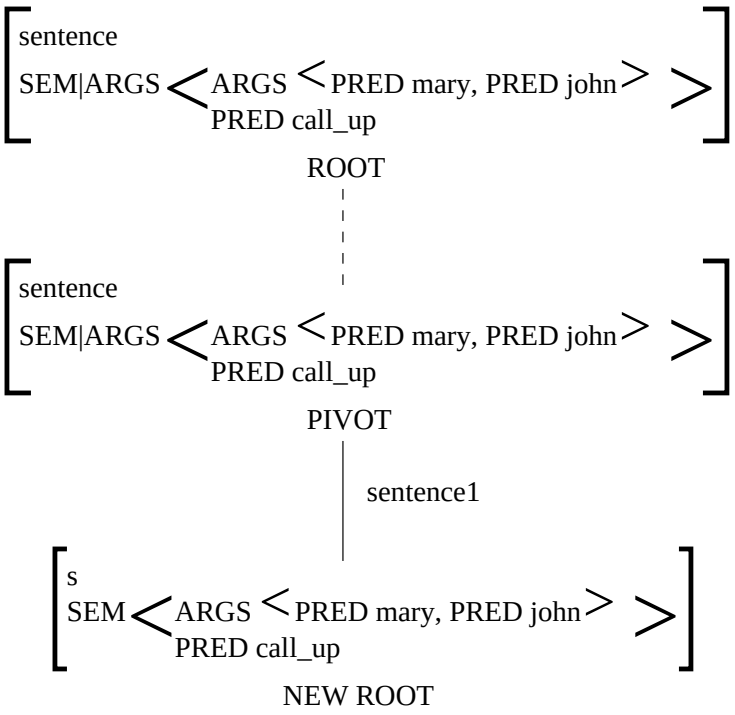

Figure 5.3: The first pivot is found.

```
cat> (s,form:finite,
        sem:S).
```

We can tell that `sentence1` is a non-chain rule because it lacks a `sem_head>` daughter, unlike, for example, the chain rule `s`:

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

The only daughter of `sentence1` becomes the new root.

The pivot chosen for that root is the lexical entry for `calls`, which is obtained by applying the lexical rule, `sg3`, to the first entry for `call` in the grammar. That pivot has no daughters, so it must now be connected by chain rules to the new root in Figure 5.3. The chain rule, `vp1`, is chosen, its semantic head is unified with the entry for `calls`, its one non-head daughter is recursively generated (which simply succeeds by unifying with the lexical entry for `john`), and its mother becomes the new chain node (Figure 5.4).

Again (Figure 5.5), chain rule `vp1` is chosen, its semantic head is unified with the new pivot, its non-head daughter is recursively generated by unifying with the lexical entry for `up`, and its mother becomes the next chain node.

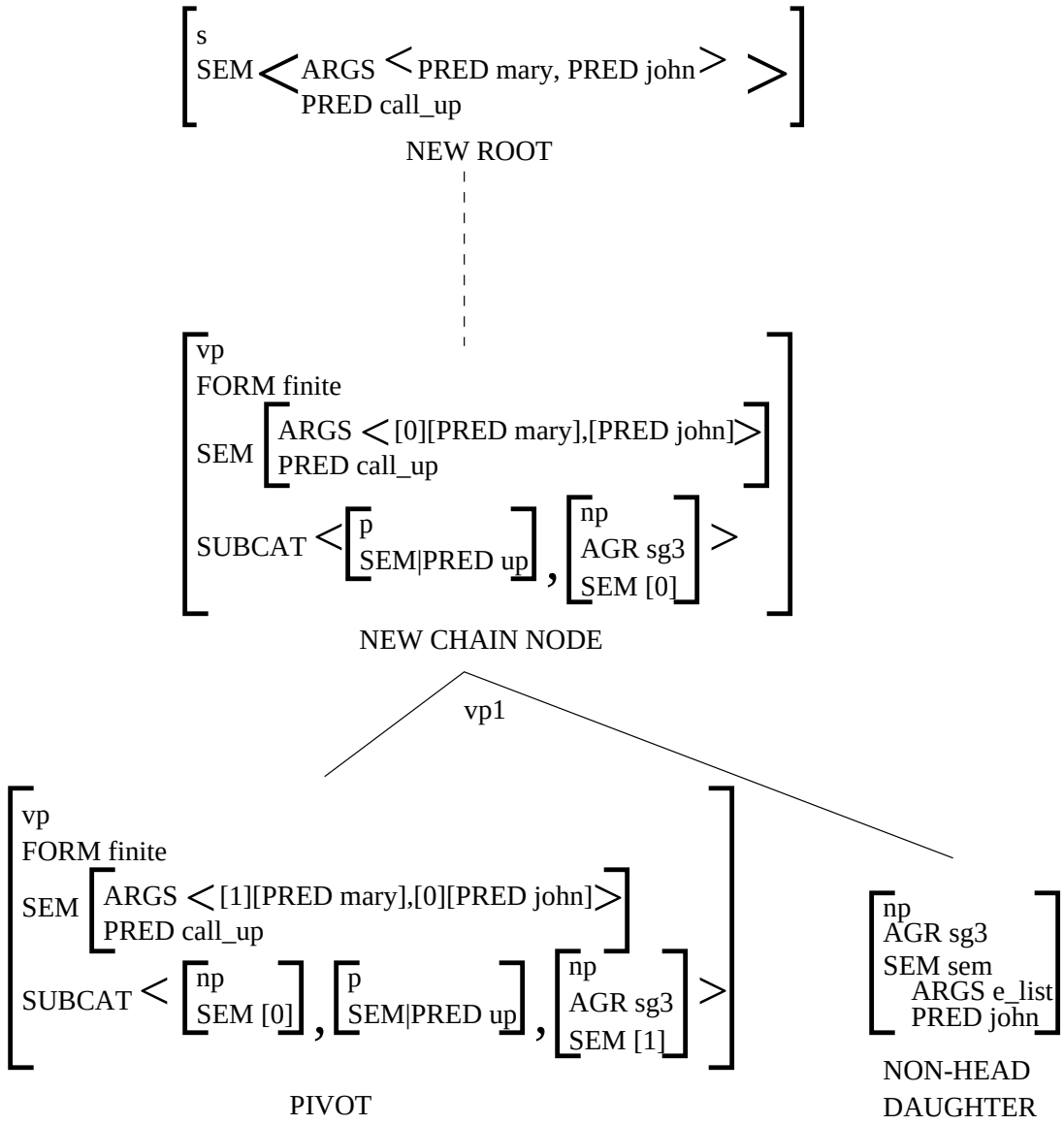

Figure 5.4: First chain rule application.

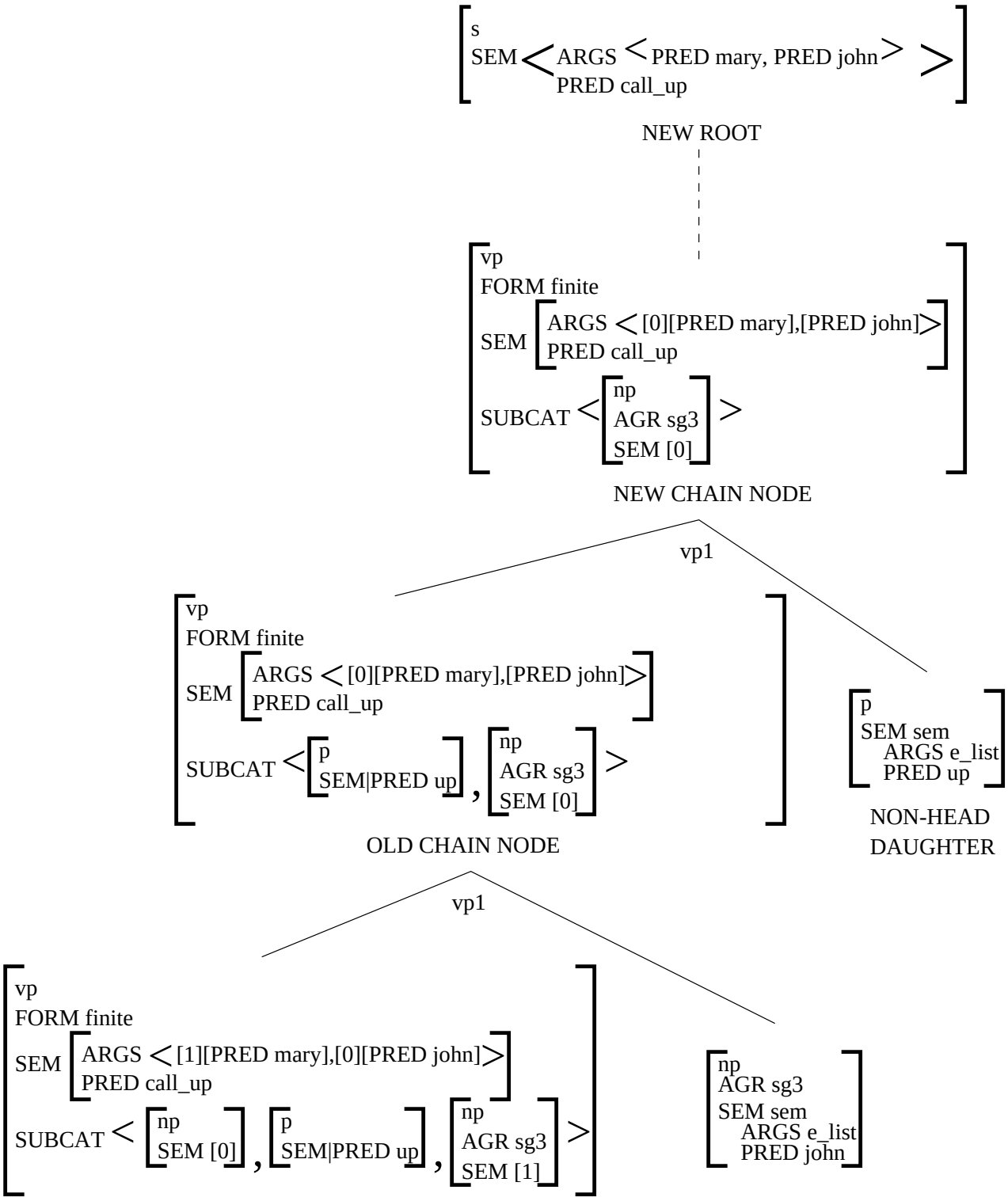

Figure 5.5: Second chain rule application.

Finally, the chain rule, `s` is chosen. Its non-head daughter is recursively generated by unifying with the lexical entry for `mary`, and its mother, the new chain node, unifies with the new root (Figure 5.6). With generation below the pivot of Figure 5.3 having been completed, it is linked to its root directly by unification, yielding the solution, `mary calls john up`.

Pivot Checking

ALE's generator uses a simple depth-first search strategy, displaying solutions as it finds them. As a result, it is not complete. Following the suggestion made in Shieber et al. (1990), ALE also checks whether there are *semantic head* → *mother* sequences that could possibly link a potential pivot to the current root before recursively generating its non-head daughters. If not, then the pivot is discarded. Semantic-head-driven generators that do not prune away such bad pivots from the search tree run a greater risk of missing solutions because top-down generation of the bad pivot's non-head daughters may not terminate, even though it can never yield solutions. This check is valuable because it incorporates *syntactic* information from the mother and semantic head into the otherwise *semantic* prediction step.

It also creates another termination problem, however, namely the potential for infinitely long *semantic head* → *mother* sequences in some grammars. To avoid this, ALE requires the user to specify a bound on the length of chain rule sequences at compile-time. This can be specified with the declaration:

```
:- chain_length(4).
```

Other values than `4` can be used, including 0. The default value is 4. ALE compiles chains of semantic head and mother descriptions of this length to perform the pivot check more efficiently at run-time.

The `sem_goal>` Operator

For the most part, the generator treats procedural attachments as the parser does — it evaluates them with respect to other daughter specifications in the order given. The one exception to this is `sem_goal>` attachments. These goals are distinguished as attached to the semantic head, and are therefore evaluated either immediately before or immediately after the `sem_head>` description. As a result, `sem_goal>` specification can only occur immediately before or immediately after a `sem_head>` specification; and thus only in chain rules. `sem_goal>` attachments are not evaluated during the pivot check described above — only the `sem_head>` and mother descriptions. During parsing, `sem_goal>` specifications are treated exactly the same as `goal>` specifications, i.e., evaluated in order.

To summarize, the order of execution for a non-chain rule specification during generation is:

- the mother description, then
- the `cat>`, `cats>`, and `goal>` descriptions, in order, from left to right.

There are no `sem_head>` or `sem_goal>` specifications in a non-chain rule. The order for a chain rule specification during generation is:

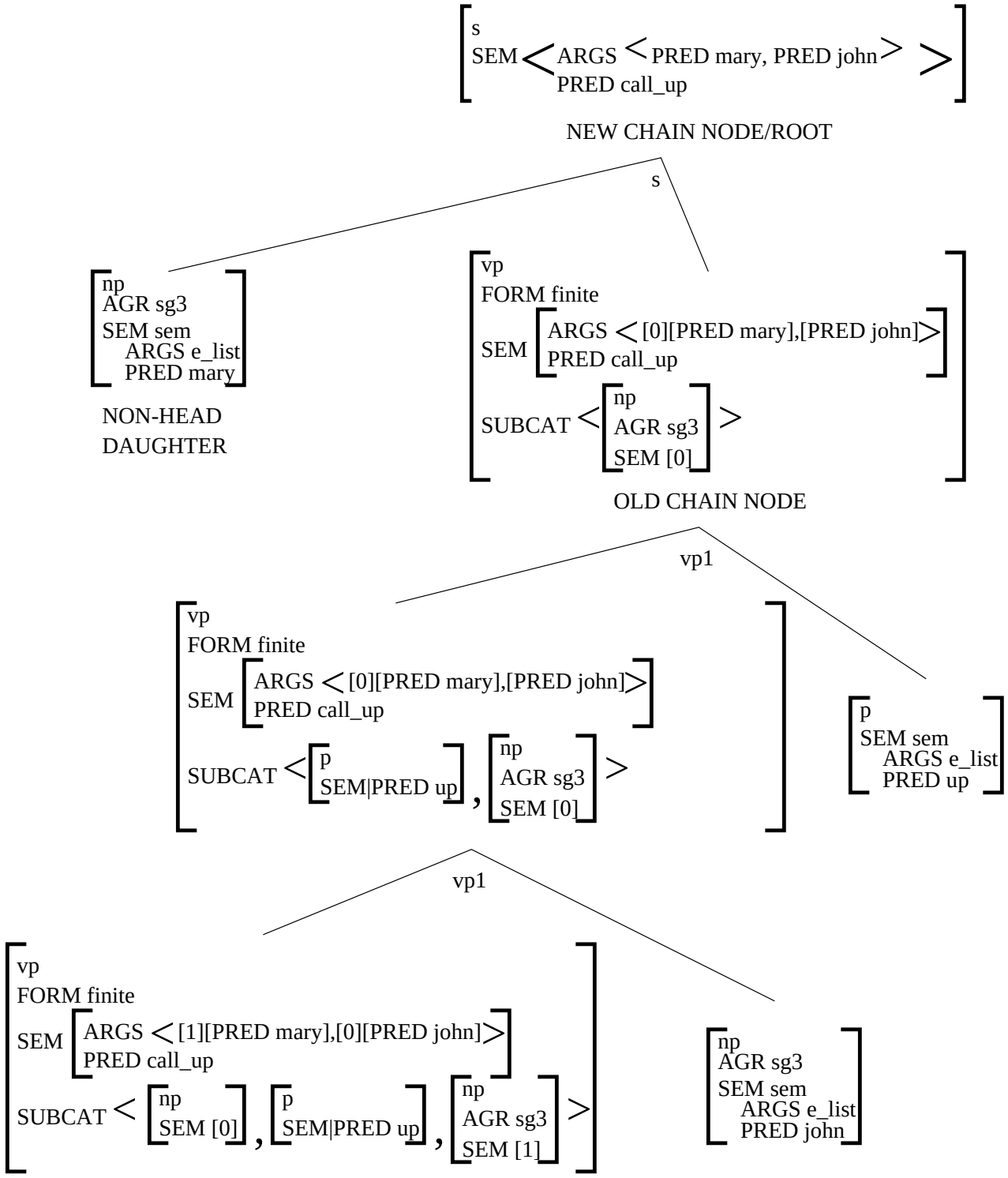

Figure 5.6: Third chain rule application and unification.

- the pre-head `sem_goal>` specification, if it exists,
- the `sem_head>` description,
- the post-head `sem_goal>` specification, if it exists,
- the mother description, then[4]
- the `cat>`, `cats>` and `goal>` specifications, in order, from left to right.

Again, practical grammar implementations will arrange information in rules in such a way as to ensure termination and to force failure as early as possible. For non-chain rules, this means making the mother and early daughters or goals as informative as possible at the description level (that is, up to where type inferencing can take over). For chain rules, the semantic head and its attachments should be maximally informative.

[4]The standard head-driven-generation algorithm enforces the mother description after the non-semantic-head-related daughters. We deviate from this order in order to enforce the pivot check, which requires instantiating the mother, more efficiently.

Chapter 6

Compiling ALE Programs

This section is devoted to showing how ALE programs can actually be compiled. ALE was developed to be run with a Prolog compiler, such as SICStus Prolog's. An SWI port of ALE is available, which also has a less extensive compilation phase. We strongly recommend SICStus Prolog. SWI Prolog does not scale up well to large-sized grammars. The local systems administrator should be able to provide help on running Prolog. This documentation only assumes the user has figured out how to run Prolog as well as write and edit files. It is otherwise self-contained.

6.1 File Management

After starting up Prolog, the following command should be used to load the ALE system:

```
| ?- compile(AleFile).
```

where *AleFile* is an atom specifying the file name in which ALE resides. For instance, in Unix, you might need to use something like: `compile('/users/carp/Prolog/ALE/ale.pl').`, or a local abbreviation for it like `compile(ale).` if the system is in a file named `ale.pl` in the local directory (SICStus and SWI can fill in the "`.pl`" suffix). With SWI Prolog, the command:

```
| ?- consult(AleFile).
```

must be used instead. Note that the argument to `compile` must be an atom, which means it should be single-quoted if it is not otherwise an atom. After the system has compiled, you should see another Prolog prompt. It is necessary to have write permission in the directory from which Prolog is invoked, because ALE creates files during compilation. But note that neither the grammar nor ALE need to be locally defined; it is only necessary to have local write permission.

ALE source code, being a kind of Prolog code, must be organized so that predicate definitions are not spread across files, unless the appropriate `multifile` declarations are made. For instance, the `sub/intro` clauses specifying the type hierarchy must all be in one file. Similarly, the definite clauses must all be in one file, as must the grammar rules and macros.

6.2 Compiling Programs

ALE can compile a program incrementally to some extent. In particular, the compiler is broken down into six primary components for compiling the type hierarchy, functional descriptions, type constraints, the attribute-value logic, the definite clauses and the grammar. Compiling the type hierarchy consists of compiling type subsumption, type unification, appropriateness specifications, and extensionality information. The logic compiler compiles predicates which know how to add a type to a feature structure, how to find a feature value in a type and how to perform feature structure unification, as well as the most general satisfiers of every type, with code attached to enforce `cons/2` constraints. Compiling the grammar consists of compiling the lexicon, empty categories, rules and lexical rules, and if compilation for generation is enabled, the `semantics/1` directive. Macros are not compiled, but are rather interpreted during compilation.

There is one predicate `compile_gram/1` that can be used to compile a whole ALE grammar from one file, as follows:

```
| ?- compile_gram(GramFile).
```

where *GramFile* is the name of the file in which the grammar resides. The compiler will display error messages to the screen when it is compiling. But since ALE uses the Prolog compiler to read the files, Prolog might also complain about syntax errors in specifying the files. In either case, there should be some indication of what the error is and which clause of the file contained it.

ALE's compiler creates code for parsing, generation, or both. As of the present version of ALE, only one grammar can be used, even if code for both modes is to be created. Two files, `ale_parse.pl` and `ale_gen.pl`, are included with the distribution, which provide some example glue code to link together two ALE processes running under SICStus Prolog 3.0 or higher in order to parse and generate with two different grammars.

At startup, ALE produces code only for parsing. To produce code for generation only, use the command:

```
| ?- generate.

compiler will produce code for generation only

yes
| ?-
```

To produce code for both parsing and generation, use `parse_and_gen` instead. To switch back to producing code for parsing only, use `parse`. Note that these commands modify the behaviour of the compiler, not the compiled code, so grammars may need to be recompiled after these directives are issued.

The following predicates are available to compile grammars and their component parts. They are listed hierarchically, with each command calling all those listed under it. Each higher-level command is nothing more than the combination of those commands below it.

Command	Requires	File	Mode	Clause
compile_gram	nothing	*	both	
compile_sig	nothing	*	both	
compile_sub_type	nothing	*		sub
compile_unify_type	compile_sub_type			
compile_approp	compile_unify_type	*		intro
compile_extensional	compile_approp	*		ext
compile_fun	compile_sig	*	both	+++>
compile_cons	compile_fun	*	both	cons
compile_logic	compile_sig		both	
compile_mgsat	compile_sig			
compile_add_to_type	compile_sig			
compile_featval	compile_add_to_type			
compile_u	compile_sig			
compile_subsume	compile_sig		parse/subtest	
compile_dcs	compile_logic	*	both	if
compile_grammar	compile_logic	*		
compile_lex_rules	compile_logic	*	parse	**>
compile_lex	compile_logic	*	parse	--->
compile_rules	compile_logic	*	parse	===>,empty
	compile_logic	*	gen	===>,empty
				--->,**>
compile_generate	compile_rules	*	gen	semantics

The table above lists which compilations must have already been compiled before the next stage of compilation can begin. Thus before `compile_grammar` can be called, `compile_logic` must be called (or equivalently, the sequence of `compile_mgsat`, `compile_add_to_type`, `compile_featval`, and `compile_u`). Each command with an asterisk in its clauses column in the above table may be given an optional file argument. The file argument should be an atom which specifies the file in which the relevant clauses can be found. The clauses needed before each stage of compilation can begin are listed to the right of the asterisks. For instance, the `if` clauses must be loaded before `compile_dcs` is called. But note that `compile_unify_type` does not require any clauses to be loaded, as it uses the compiled definition of `sub_type` rather than the user specification in its operation. Thus changes to the signature in the source file, even if the source file is recompiled, will not be reflected in `compile_unify_type` if they have not been recompiled by `compile_sub_type` first. If an attempt is made to compile a part of a program where the relevant clauses have not been asserted, an error will result.

Note that `compile_subsume` only compiles code if subsumption checking (p. 93) and parsing have been enabled.

Each of the lowest level commands generates intermediate Prolog source code for that function, which is then compiled further by a Prolog compiler. ALE uses a term-expansion-based compiler in both SICStus and SWI Prologs that avoids the necessity for creating intermediate files. It also improves the speed of intermediate code compilation. Because both SICStus and SWI Prologs require the user to read a file on disk in order to use their compilers, ALE must create a zero-byte file called

`.alec_throw` to throw control to its intermediate code compiler. For that reason, the Prolog process must have write permission in the local directory to create this file, if it does not already exist.

After a grammar is compiled, the system plus grammar code can be saved with the command:

```
| ?- save_program(File).
```

This will save the state of the Prolog database in *File*. SICStus users should normally use this rather than `save/1`, which creates a larger file by saving other information like the state of Prolog's internal stacks. The SWI Prolog command is `qsave_program(File)`. With either Prolog, the state can be reloaded, by executing the saved file directly.

In general, whenever the ALE source program is changed, it should be recompiled from the point of change. For instance, if the definite clauses are the only thing that have changed since the last compilation, then only `compile_dcs(`*FileSpec*`)` needs to be run. But if in changing the definite clauses, the type hierarchy had to be changed, then everything must be recompiled.

ALE treats lexicon compilation differently than the other stages. Two commands, `lex_compile/0` and `lex_consult/0`, control whether the intermediate code for the lexicon and empty categories is compiled or consulted (a lesser degree of compilation). Lexicon compilation is usually the most time-consuming stage of grammar compilation in ALE, and consulting the code for this stage can result in a substantial compile-time speed-up. The decrease in run-time performance is only significant in grammars with a high degree of lexical ambiguity, i.e., where one string has a very large number of entries in the lexicon. By default, ALE consults the code for the lexicon. In SWI Prolog, only lexicon consulting is available.

When consulting is chosen, the lexicon and empty categories are also consulted *dynamically*. This means that individual entries can be retracted and added without recompiling the entire lexicon. To retract a lexical entry, use the command:

```
| ?- retract_lex(LexSpec).
```

LexSpec can either be a word, or a list of words. The words given to `retract_lex/1` are not closed under `morph` rules — derived entries with different forms must be retracted explicitly. `retract_lex/1` iterates through all of the entries that match the given word(s), asking for each one whether it should be retracted. `retractall_lex/1` will remove all of them without asking.

To add lexical entries, use the command:

```
| ?- update_lex(UpdateFile).
```

UpdateFile is a file containing new lexical entries and empty categories. New lexical entries are closed under lexical rules, as usual.[1]

[1] Earlier versions of ALE also permitted incremental updating and retraction of empty categories. Because empty categories are now closed under phrase structure rules at compile-time, this is no longer possible.

6.3 Compile-Time Error Messages

There are three sources of compile-time messages generated by ALE: Prolog messages, ALE errors, and ALE warnings.

ALE uses Prolog term input and output, thus requiring the input to be specified as a valid Prolog program. Of course, any ALE program meeting the ALE syntax specification will not cause Prolog errors. If there is a Prolog error generated, there is a corresponding bug in the grammar file(s). Prolog error messages usually generate a message indicating what kind of error it found, and just as importantly, which line(s) of the input the error was found in. The most common Prolog error messages concern missing periods or operators which cannot be parsed. Such errors are usually caused by bad punctuation such as missing periods, misplaced commas, commas before semicolons in disjunctions, etc. These errors are usually easy to track down.

Prolog also generates warnings in some circumstances. In particular, if you only use a variable once in a definition, it will report a singleton variable warning. The reason for this is that variables that only occur once are useless in that they do not enforce any structure sharing. There is little use for singleton variables in ALE outside of the Prolog goals in morphological rules and some macro parameters. Usually a singleton variable indicates a typing error, such as typing `AgrNum` in one location and `Agrnum` in another. It is standard Prolog practice to replace all singleton variables with *anonymous* variables. An anonymous variable is a variable which begins with the underscore character. For instance, a singleton variable such as `Head` can be replaced with the anonymous variable `_Head`, or even just `_`, to suppress such singleton variable warnings. Two occurrences of the simple anonymous variable `_` are not taken to be co-referential, but two occurrences of something like `_Head` are taken to be co-referential. In particular, the two descriptions, `(foo:X, bar:X)` and `(foo:_X, bar:_X)` are equivalent to each other, but distinct from `(foo:_,bar:_)` in that the latter description does not indicate any structure sharing. The second description above is considered bad style, though, as it uses the anonymous variable `_X` co-referentially.

Besides Prolog syntax errors, there are many errors that ALE is able to detect at compile time. These errors will be flagged during compilation. Most errors give some indication of the program clause in which they are found. Some errors may be serious enough to halt compilation before it is finished. In general, it is a good idea to fix all of the errors before trying to run a program, as the error messages only report serious bugs in the code, such as type mismatches, unspecified types, ill-formed rules, etc.

In certain cases, it is preferable to disable those error messages concerned with ALE's inability to add incompatible descriptions to a feature structure. This is especially true during lexicon and empty category compilation, when, due to the interaction of disjunctions and type constraints, the number of such errors can be overwhelming. In the current version of ALE, these errors are automatically disabled during lexicon and empty category compilation, and enabled otherwise. Commands will be added to future versions so that the user may control when these errors should be displayed.

Less serious problems are flagged with warning messages. Warning messages do not indicate an error, but may indicate an omission or less than optimal ALE

programming style.

The ALE error and warning messages are listed in an appendix at the end of this report, along with an explanation. The manual for the Prolog in which ALE is being run in will probably list the kinds of errors generated by the Prolog compiler.

Chapter 7

Running and Debugging ALE Programs

After the ALE program compiles without any error messages, it is possible to test the program to make sure it does what it is supposed to. We consider the problem from the bottom-up, as this is the best way to proceed in testing grammars. ALE does not have a sophisticated input/output package, and thus all ALE procedures must be accessed through Prolog queries.

7.1 Testing the Signature

Once the signature is compiled, it is possible to test the results of the compilation. To test whether or not a type exists, use the following query:

```
| ?- type(Type).

Type = bot ?;

Type = cat ?;

Type = synsem ?

yes
```

Note that the prompt `| ?-` is provided by Prolog, while the query consists of the string `type(Type).`, including the period and a return after the period. Prolog then responds with instantiations of any variables in the query if the query is successful. Thus the first solution for `Type` that is found above is `Type = bot`. After providing an instantiation representing the solution to the query, Prolog then provides another prompt, this time in the form of a single question mark. After the first prompt above, the user typed a semicolon and return, indicating that another solution is desired. The second solution Prolog found was `Type = cat`. After this prompt, the user requested a third solution. After the third solution, `Type = synsem`, the user simply input a return, indicating that no more solutions were desired. These two options, semicolon followed by return, and a simple return, are the only ones relevant for ALE. If the anonymous variable `_` is used in a query, no substitutions

are given for it in the solution. If there are no solutions to a query, Prolog returns `no` as an answer. Consider the following two queries:

```
| ?- type(bot).

yes

| ?- type(foobar).

no
```

In both cases, no variables are given in the input, so a simple `yes/no` answer, followed by another prompt, is all that is returned.

The second useful probe on the signature indicates type subsumptions and type unifications. To test type subsumption, use the following form of query:

```
| ?- sub_type(X,Y).

X = and,
Y = and ?;

X = backward,
Y = backward ?

yes
```

Note that with two variables, substitutions for both are given, allowing the possibility of iterating through the cases. In general, wherever a variable may be used in a query, a constant may also be used. Thus `sub_type(synsem,forward).` is a valid query, as are `sub_type(synsem,X)` and `sub_type(Y,forward)`. The first argument is the more general type, with the second argument being the subtype.

Type unifications are handled by the following form of query:

```
| ?- unify_type(T1,T2,T).
```

The interpretation here is that `T1` unified with `T2` produces `T3`. As before, any subset of the three variables may be instantiated for the test and the remaining variables will be solved for.

The following query will indicate whether given features have been defined and can also be used to iterate through the features if the argument is uninstantiated:

```
| ?- feature(F).
```

Feature introduction can be tested by:

```
| ?- introduce(F,T).
```

which holds if feature `F` is introduced at type `T`.

Type constraints can be tested using:

```
| ?- show_cons(Type).
```

which will display the description of the constraint assigned to the type, Type.

Finally, the inherited appropriateness function can be tested by:

```
| ?- approp(Feat,Type,Restr).
```

A solution indicates that the value for feature `Feat` for a type `Type` structure is of type `Restr`. As usual, any of the variables may be instantiated, so that it is possible to iterate through the types appropriate for a given feature or the features appropriate for a given type, the restrictions on a given feature in a fixed type, and so on.

There is one higher-level debugging routine for the signature that outputs a complete specification for a type, including a list of its subtypes and supertypes, along with the most general feature structure of that type (after all type inference and constraint satisfaction has been performed). An example of the `show_type/1` query is as follows:

```
| ?- show_type functional.

TYPE: functional
SUBTYPES: [forward,backward]
SUPERTYPES: [synsem]
MOST GENERAL SATISFIER:
     functional
     ARG synsem
     RES synsem
```

If `synsem` had any appropriate features, these would have been added, along with their most general appropriate values.

7.2 Evaluating Descriptions

Descriptions can be evaluated in order to find their most general satisfiers. ALE provides the following form of query:

```
| ?- mgsat tl:e_list.

ne_list_quant
HD quant
   RESTR proposition
   SCOPE proposition
   VAR individual
TL e_list

ANOTHER?  n.

yes
```

Note that there must be whitespace between the `mgsat` and the description to be satisfied. The answer given above is the most general satisfier of the description `tl:e_list` using the signature in the categorial grammar in the appendix. It is

important to note here that type inference is being performed to find most general satisfiers. In the case at hand, because lists in the categorial grammar are typed to have quantifiers as their HD values, the value of the HD feature in the most general satisfier has been coerced to be a quantifier.

Satisfiable non-disjunctive descriptions always have unique most general satisfiers as a consequence of the way in which the type system is constrained. But a description with disjunctions in it may have multiple satisfiers. Consider the following query:

```
| ?- mgsat hit,hitter:(j;m).

hit
HITTEE individual
HITTER j

ANOTHER?  y.

hit
HITTEE individual
HITTER m

ANOTHER?  y.

no
```

After finding the first most general satisfier to the description, the user is prompted as to whether or not another most general satisfier should be sought. As there are only two most general satisfiers of the description, the first request for another satisfier succeeds, while the second one fails. Failure to find additional solutions is indicated by the no response from Prolog.

Error messages will result if there is a violation of the type hierarchy in the query. For instance, consider the following query containing two type errors before a satisfiable disjunct:

```
| ?- mgsat hd:j ; a ; j.

add_to could not add incompatible type j to:
     quant
     RESTR proposition
     SCOPE proposition
     VAR individual

add_to could not add undefined type: a to
     bot

MOST GENERAL SATISFIER OF: hd:j;a;j

j
```

```
ANOTHER?
```

Here the two errors are indicated, followed by a display of the unique most general satisfiers. The problem with the first disjunct is that lists have elements which must be of the quantifier type, which conflicts with the individual type of `j`, while the second disjunct involves an undefined type `a`. Note that in the error messages, there is some indication of how the conflict arose as well as the current state of the structure when the error occurred. For instance, the system had already figured out that the head must be a quantifier, which it determined before arriving at the incompatible type `j`. The conflict arose when an attempt was made to add the type `j` to the `quant` type object.

To explore unification, simply use conjunction and `mgsat`. In particular, to see the unification of descriptions `D1` and `D2`, simply display the most general satisfiers of `D1`, `D2`, and their conjunction `(D1,D2)`. To obtain the correct results, `D1` and `D2` must not share any common variables. If they do, the values of these will be unified across `D1` and `D2`, a fact which is not represented by the most general satisfiers of either `D1` or `D2`. Providing most general satisfiers also allows the user to test for subsumption or logical equivalence by visual inspection, by using `mgsat/1` and comparing the set of solutions. Future releases should contain mechanisms for evaluating subsumption (entailment), and hence logical equivalence of descriptions.

`mgsat` can also be used to test functional descriptions, e.g., for the functional append on p. 30:

```
| ?- mgsat append([bot],[bot,bot]).

ne_list
HD bot
TL ne_list
  HD bot
  TL ne_list
    HD bot
    TL e_list

ANOTHER?
```

The Prolog predicate `iso_desc/2` can be used to discover whether two descriptions evaluate to the same feature structure. This can be useful for testing extensional type declarations.

```
| ?- iso_desc(X,X).

X = _A-bot ?

yes
| ?- iso_desc((a_ atom),(a_ atom)).     % a_ atoms are extensional

yes
| ?- iso_desc(b,b).         % for b, intensional
```

```
no
| ?- iso_desc(a,a).         % for a, extensional, with feature f

no
| ?- iso_desc(f:(a_ at1),f:(a_ at1)).   % f approp. to a_ atoms

yes
| ?- iso_desc(f:(a_ at1),f:(a_ at2)).

no
```

7.3 Hiding Types and Features

With a feature structure system such as ALE, grammars and programs often manipulate very large feature structures. To aid in debugging, two queries allow the user to focus attention on particular types and features by supressing the printing of other types and features.

The following command supresses printing of a type:

```
| ?- no_write_type(T).
```

After `no_write_type(`T`)` is called, the type T will no longer be displayed during printing. To restore the type T to printed status, use:

```
| ?- write_type(T).
```

If T is a variable in a call to `write_type/1`, then all types are subsequently printed. Alternatively, the following query restores printing of all types:

```
| ?- write_types.
```

Features and their associated values can be supressed in much the same way as types. In particular, the following command blocks the feature F and its values from being printed:

```
| ?- no_write_feat(F).
```

To restore printing of feature F, use:

```
| ?- write_feat(F).
```

If F is a variable here, all features will subsequently be printed. The following special query also restores printing of all features.

```
| ?- write_feats.
```

7.4 Evaluating Definite Clause Queries

It is possible to display definite clauses in feature structure format by name. The following form of query can be used:

```
| ?- show_clause append.

HEAD: append(e_list,
             [0] bot,
             [0] )
BODY: true

ANOTHER?  y.

HEAD: append(ne_list_quant
             HD [0] quant
                RESTR proposition
                SCOPE proposition
                VAR individual
             TL [1] list_quant,
             [2] bot,
             ne_list_quant
             HD [0]
             TL [3] list_quant)
BODY: append([1],
             [2],
             [3])

ANOTHER?  y.

no
```

Note that this example comes from the categorial grammar in the appendix. Also note that the feature structures are displayed in full with tags indicating structure sharing. Next, note that prompts allow the user to iterate through all the clauses. The number of solutions might not correspond to the number of clause definitions in the program due to disjunctions in descriptions which are resolved non-deterministically when displaying rules. But it is important to keep in mind that this feature structure notation for rules is not the one ALE uses internally, which compiles rules down into elementary operations which are then compiled, rather than evaluating them as feature structures by unification. In this way, ALE is more like a logic programming compiler than an interpreter. Finally, note that the arity of the predicate being listed may be represented in the query as in Prolog. For instance, the query `show_clause append/3` would show the clauses for `append` with three arguments.

Definite clauses in ALE can be evaluated by using a query such as:

```
| ?- query append(X,Y,[a,b]).

append(e_list,
       [0] ne_list
       HD a
       TL ne_list
          HD b
```

```
          TL e_list,
       [0] )

ANOTHER?  y.
append(ne_list
       HD [0] a
       TL e_list,
       [1] ne_list
       HD b
       TL e_list,
       ne_list
       HD [0]
       TL [1] )

ANOTHER?  y.
append(ne_list
       HD [0] a
       TL ne_list
          HD [1] b
          TL e_list,
       [2] e_list,
       ne_list
       HD [0]
       TL ne_list
          HD [1]
          TL [2] )

ANOTHER?  y.

no
```

The definition of `append/3` is taken from the syllabification grammar in the appendix. After displaying the first solution, ALE queries the user as to whether or not to display another solution. In this case, there are only three solutions, so the third query for another solution fails. Note that the answers are given in feature structure notation, where the macro `[a,b]` is converted to a head/tail feature structure encoding.

Unlike Prolog, in which a solution is displayed as a substitution for the variables in the query, ALE displays a solution as a satisfier of the entire query. The reason for this is that structures which are not given as variables may also be further instantiated due to the type system. Definite clause resolution in ALE is such that only the most general solutions to queries are displayed. For instance, consider the following query, also from the syllabification grammar in the appendix:

```
| ?- query less_sonorous(X,r).

less_sonorous(nasal,
              r)
```

```
ANOTHER?  y.

less_sonorous(sibilant,
              r)

ANOTHER?  n.
```

Rather than enumerating all of the `nasal` and `sibilant` types, ALE simply displays their supertype. On the other hand, it is important to note that the query `less_sonorous(s,r)` would succeed because `s` is a subtype of `sibilant`. This example also clearly illustrates how ALE begins each argument on its own line arranged with the query.

In general, the goal to be solved must be a literal, consisting only of a relation applied to arguments. In particular, it is not allowed to contain conjunction, disjunction, cuts, or other definite clause control structures. To solve a more complex goal, a definite clause must be defined with the complex goal as a body and then the head literal solved, which will involve the resolution of the body.

There are no routines to trace the execution of definite clauses. Future releases of ALE will contain a box port tracer similar to that used for Prolog. At present, the best suggestion is to develop definite clauses modularly and test them from the bottom-up to make sure they work before trying to incorporate them into larger programs.

7.5 Displaying Grammars

ALE provides a number of routines for displaying and debugging grammar specifications. After compile-time errors have been taken care of, the queries described in this section can display the result of compilation.

Lexical entries can be displayed using the following form of query:

```
| ?- lex(kid).

WORD: kid
ENTRY:
cat
QSTORE e_list
SYNSEM basic
       SEM property
           BODY kid
                ARG1 [0] individual
           IND [0]
       SYN n

ANOTHER?  y.

no
```

As usual, if there are multiple entries, ALE makes a query as to whether more should be displayed. In this case, there was only one entry for `kid` in the categorial grammar

in the appendix.

Another predicate, `export_words(`*Stream*`,`*Delimiter*`)`, writes an alphabetised list of all of the words in the lexicon, separated by *Delimiter*, to *Stream*. In SICStus Prolog, for example, `export_words(user_output,'\n')` will write the words to standard output (such as the screen), one to a line.

Empty lexical entries can be displayed using:

```
| ?- empty.

EMPTY CATEGORY:
    cat
    QSTORE ne_list_quant
           HD some
              RESTR [0] proposition
              SCOPE proposition
              VAR [1] individual
           TL e_list
    SYNSEM forward
           ARG basic
               SEM property
                   BODY [0]
                   IND [1]
               SYN n
           RES basic
               SEM [1]
               SYN np

ANOTHER?  no.
```

Note that the number specification was removed to allow the empty category to be processed with respect to the categorial grammar type system. As with the other display predicates, `empty` provides the option of iterating through all of the possibilities for empty categories.

Grammar rules can be displayed by name, as in:

```
| ?- rule forward_application.

RULE: forward_application

MOTHER:

  cat
  QSTORE [4] list_quant
  SYNSEM [0] synsem

DAUGHTERS/GOALS:

CAT  cat
     QSTORE [2] list_quant
```

```
     SYNSEM forward
            ARG [1] synsem
            RES [0]

CAT  cat
     QSTORE [3] list_quant
     SYNSEM [1]

GOAL  append([2],
             [3],
             [4])

ANOTHER?  n.
```

Rules are displayed as most general satisfiers of their mother, category and goal descriptions. It is important to note that this is for display purposes only. The rules are not converted to feature structures internally, but rather to predicates consisting of low-level compiled instructions. Displaying a rule will also flag any errors in finding most general satisfiers of the categories and rules in goals, and can thus be used for rule debugging. This can detect errors not found at compile-time, as there is no satisfiability checking of rules performed during compilation.

Macros can also be displayed by name, using:

```
| ?- macro np(X).

MACRO:
    np([0] sem_obj)
ABBREVIATES:
    basic
    SEM [0]
    SYN np

ANOTHER?  n.
```

First note that the macro name itself is displayed, with all descriptions in the macro name given replaced with their most general satisfiers. Following the macro name is the macro satisfied by the macro description with the variables instantiated as shown in the macro name display. Note that there is sharing between the description in the macro name and the `SEM` feature in the result. This shows where the parameter is added to the macro's description.

Finally, it is possible to display lexical rules, using the following query:

```
| ?- lex_rule plural_n.

LEX RULE: plural_n
INPUT CATEGORY:
    n
    NUM sing
    PERS pers
```

```
  OUTPUT CATEGORY:
      n
      NUM plu
      PERS pers
  MORPHS:
      [g,o,o,s,e] becomes [g,e,e,s,e]
      [k,e,y] becomes [k,e,y,s]
      A,[m,a,n] becomes A,[m,e,n]
      A,B becomes A,B,[e,s]
          when fricative(B)
      A,[e,y] becomes A,[i,e,s]
      A becomes A,[s]

  ANOTHER?  n.
```

Note that the morphological components of a rule is displayed in canonical form when it is displayed. Note that variables in morphological rules are displayed as upper case characters. When there is sharing of structure between the input and output of a lexical rule, it will be displayed as such. As with the other ALE grammar display predicates, if there are multiple solutions to the descriptions, these will be displayed in order. Also, if there is a condition on the categories in the form of an ALE definite clause goal, this condition will be displayed before the morphological clauses. As with grammar rules, lexical rules are compiled internally and not actually executed as feature structures. The feature structure notation is only for display. Also, as with grammar rules, displaying a lexical rule may uncover inconsistencies which are not found at compile time.

7.6 Executing Grammars: Parsing

In this section, we consider the execution of ALE phrase structure grammars compiled for parsing. The examples shown in this section have been produced while running with the mini-interpreter off. The mini-interpreter will be discussed in the next section.

The primary predicate for parsing is illustrated as follows:

```
| ?- rec [john,hits,every,toy].

STRING:
0 john 1 hits 2 every 3 toy 4

CATEGORY:
cat
QSTORE e_list
SYNSEM basic
       SEM every
           RESTR toy
                 ARG1 [0] individual
           SCOPE hit
```

```
                HITTEE [0]
                HITTER j
            VAR [0]
       SYN s

ANOTHER?  y.

CATEGORY:
cat
QSTORE ne_list_quant
       HD every
          RESTR toy
                ARG1 [0] individual
          SCOPE proposition
          VAR [0]
       TL e_list
SYNSEM basic
       SEM hit
           HITTEE [0]
           HITTER j
       SYN s

ANOTHER?  y.

no
```

The first thing to note here is that the input string must be entered as a Prolog list of atoms. In particular, it must have an opening and closing bracket, with words separated by commas. No variables should occur in the query, nor anything other than atoms. The first part of the output repeats the input string, separated by numbers (*nodes* in the chart) which indicate positions in the string for later use in inspecting the chart directly. ALE asserts one lexical item for every unit interval, with empty categories being stored as loops from every single node to itself. The second part of the output is a category which is derived for the input string. If there are multiple solutions, these can be iterated through by providing positive answers to the query. The final `no` response above indicates that the category displayed is the only one that was found. If there are no parses for a string, an answer of `no` is returned, as with:

```
| ?- rec([runs,john]).

STRING:
0 runs 1 john 2

no
```

Notice that there is no notion of "distinguished start symbol" in parsing. Rather, the recognizer generates all categories that it can find for the input string. This allows sentence fragments and phrases to be analyzed, as in:

```
| ?- rec [big,kid].

STRING:
0 big 1 kid 2

CATEGORY:
cat
QSTORE ne_list_quant
       HD some
          RESTR and
                CONJ1 kid
                      ARG1 [0] individual
                CONJ2 big
                      ARG1 [0]
          SCOPE proposition
          VAR [0]
       TL e_list
SYNSEM basic
       SEM [0]
       SYN np

ANOTHER?  n.
```

There is also a two-place version of `rec` that displays only those parses that satisfy a given description:

```
| ?- rec([big,kid],s).

STRING:
0 big 1 kid 2

no
```

This call to `rec/2` failed because there were no parses of `big kid` of type, `s`. Internally, the parser still generates all of the edges that it normally does — the extra description is only applied at the end as a filter.

Once parsing has taken place for a sentence using `rec/1`, it is possible to look at categories that were generated internally. In general, the parser will find every possible analysis of every substring of the input string, and these will be available for later inspection. For instance, suppose the last call to `rec/1` executed was `rec [john,hits,every,toy]`, the results of which are given above. Then the following query can be made:

```
| ?- edge(2,4).

COMPLETED CATEGORIES SPANNING: every toy

cat
QSTORE ne_list_quant
```

```
        HD every
           RESTR toy
                 ARG1 [0] individual
           SCOPE proposition
           VAR [0]
        TL e_list
  SYNSEM basic
        SEM [0]
        SYN np

  Edge created for category above:
       index: 20
        from: 2 to: 4
      string: every toy
        rule:  np_det_nbar
   # of dtrs: 2
  Action(retract,dtr-#,continue,abort)?
  |: continue.

no
| ?-
```

The possible replies in the action-line will be discussed in the next section. This tells us that from positions `2` to `4`, which covers the string `every toy` in the input, the indicated category was found. Even though an active chart parser is used, it is not possible to inspect active edges. This is because ALE represents active edges as dynamic structures that are not available after they have been evaluated.

Using `edge/2` it is possible to debug grammars by seeing how far analyses progressed and by inspecting analyses of substrings.

There is also a predicate, `rec/4` that binds the answer to variables instead of displaying it:[1]

```
  | ?- rec([kim,walks],Ref,SVs,Iqs).

  Iqs = [ineq(_A,index(_B-third,_C-plur,_D-masc),_E,index(_B-third,_C-plur,
  _D-masc),done)],
  SVs =  phrase(_F-synsem(_G-loc(_H-cat(_I-verb(_J-minus,_K-minus,_L-none,_M-
  bool,_N-fin),_O-unmarked,_P-e_list),...)))?
```

`Ref` is an unbound variable that represents the root node of the resulting feature structure. `SVs` is a term whose functor is the type of the feature structure, whose arity is the number of appropriate features for that type, and whose arguments are the values at those appropriate features in alphabetical order. Each value, in turn, is of the form, `Ref-SVs`, another pair of root variable and type-functored term. `Iqs` is a list of the inequations that apply to the feature structure and its substructures. Each member of the list represents a disjunction of inequations, i.e., one must be

[1] Actually, this example is a bit of an improvisation — the sample HPSG grammar included in the ALE distribution does not use inequations.

satisfied, with the list itself being interpreted conjunctively, i.e., every member must be satisfied. Each member is represented by a chain of `ineq/5` structures:

```
ineq(Ref1,SVs1,Ref2,SVs2,ineq(.....,done)...)
```

The first four arguments represent the `Ref-SVs` pairs of the two inequated feature structures of the first disjunct. The fifth argument contains another `ineq/5` structure, or `done/0`. These three structures are suitable for passing to `gen/3` or `gen/4`. These representations are not grounded; so if you want to assert them into a database, be sure to assert them all in one predicate to preserve variable sharing. For more details on ALE's internal representation, the reader is referred to Carpenter and Penn (1996).

There is also a `rec/5`, which works just like `rec/4`, but filters its output through the description in its last argument, just like `rec/2`:

```
| ?- rec([kim,walks],Ref,SVs,Iqs,phrase).

Iqs = [ineq(_A,index(_B-third,_C-plur,_D-masc),_E,index(_B-third,_C-plur,
_D-masc),done)],
SVs =  phrase(_F-synsem(_G-loc(_H-cat(_I-verb(_J-minus,_K-minus,_L-none,_M-
bool,_N-fin),_O-unmarked,_P-e_list),...)))?
```

The call succeeds here because the given answer is of type, `phrase`.

`rec_list/2` iteratively displays all of the solutions for each string in a list of list of words that satisfy a description:

```
| ?- rec_list([[kim,sees,sandy],[sandy,sees,kim]],phrase).

STRING:
0 kim 1 sees 2 sandy 3
CATEGORY:

phrase
QRETR e_list
QSTORE e_set
SYNSEM synsem...

ANOTHER? y.

STRING:
0 sandy 1 sees 2 kim 3
CATEGORY:

phrase
QRETR e_list
QSTORE e_set
SYNSEM synsem

ANOTHER?  y.

no
```

If no filtering through a description is desired, the description, `bot`, which is trivially satisfied, can be used. When `rec_list/2` runs out of solutions for a string, it moves on to the next string.

`rec_best/2` iteratively displays all of the solutions satisfying a given description for the *first* string in a list of list of words that has a solution satisfying that description.

```
| ?- rec_best([[kim,sees,sandy],[sandy,sees,kim]],phrase).

STRING:
0 kim 1 sees 2 sandy 3
CATEGORY:

phrase
QRETR e_list
QSTORE e_set
SYNSEM synsem...

ANOTHER? y.

no
```

When `rec_best/2` runs out of solutions for a string that had at least one solution, it fails. It only tries the strings in its first argument until it finds one that has solutions.

There is also a three-place `rec_list/3` that collects the solutions to `rec_list/2` in a list of terms of the form `fs(Tag-SVs,Iqs)`.

7.7 Executing Grammars: Generation

The generator can be used with the predicate `gen/1,3` predicate. It can take a single description argument:

```
| ?- gen((sentence,
          sem:(pred:decl,
               args:[(pred:call_up,
                      args:[pred:mary,pred:john])]))).

CATEGORY:

sentence
SEM sem
    ARGS arg_ne_list
         HD sem
            ARGS arg_ne_list
                 HD sem
                    ARGS arg_list
                    PRED mary
                 TL arg_ne_list
```

```
                    HD sem
                       ARGS arg_list
                       PRED john
                    TL e_list
               PRED call_up
          TL e_list
     PRED decl


STRING:
mary calls john up

ANOTHER?  y.

STRING:
mary calls up john

ANOTHER?  y.

no
```

Notice the extra set of parentheses necessary to make the whole description a single argument for `gen/1`.

`gen` can also take three arguments:

```
gen(Ref,SVs,Iqs)
```

where `Ref`, `SVs` and `Iqs` are the three parts of ALE's internal representation of a feature structure as defined in the last section. This alternative is most useful when the feature structure has been generated before by another process, like parsing, or retrieved from a database.

```
| ?- rec([john,calls,mary,up],Ref,SVs,Iqs),gen(Ref,SVs,Iqs).

CATEGORY:

sentence
SEM sem
    ARGS arg_ne_list
         HD sem
            ARGS arg_ne_list
                 HD sem
                    ARGS e_list
                    PRED john
                 TL arg_ne_list
                    HD sem
                       ARGS e_list
                       PRED mary
                    TL e_list
```

```
            PRED call_up
          TL e_list
    PRED decl


STRING:
john calls mary up

ANOTHER?  n.

Iqs = [],
SVs = sentence(_O-sem(_N-arg_ne_list(_M-sem(_L-arg_ne_list(_K-
sem(_J-e_list,_I-john),_H-arg_ne_list(_G-sem(_F-e_list,_E-mary),
_D-e_list)),_C-call_up),_B-e_list),_A-decl)) ?

yes
```

In both cases, ALE will print the input feature structure and then will generate and display all possible string solutions through backtracking.

It is also possible to bind the string to a variable, using `gen/4` :

```
gen(Ref,SVs,Iqs,Ws).
```

`Ws` will non-deterministically be bound to the word lists that constitute valid solutions to the generation problem. This can be used as input to `rec/1`, for example.

```
| ?- rec([john,calls,mary,up],Ref,SVs,Iqs),gen(Ref,SVs,Iqs,Ws).

Iqs = [],
SVs = sentence(_O-sem(_N-arg_ne_list(_M-sem(_L-arg_ne_list(_K-
sem(_J-e_list,_I-john),_H-arg_ne_list(_G-sem(_F-e_list,_E-mary),
_D-e_list)),_C-call_up),_B-e_list),_A-decl)),
Ws = [john,calls,mary,up] ? ;

Iqs = [],
SVs = sentence(_O-sem(_N-arg_ne_list(_M-sem(_L-arg_ne_list(_K-
sem(_J-e_list,_I-john),_H-arg_ne_list(_G-sem(_F-e_list,_E-mary),
_D-e_list)),_C-call_up),_B-e_list),_A-decl)),
Ws = [john,calls,up,mary] ? ;

Iqs = [],
SVs = s(_L-finite,_K-sem(_J-arg_ne_list(_I-sem(_H-e_list,_G-john),
_F-arg_ne_list(_E-sem(_D-e_list,_C-mary),_B-e_list)),_A-call_up)),
Ws = [john,calls,mary,up] ? ;

Iqs = [],
SVs = s(_L-finite,_K-sem(_J-arg_ne_list(_I-sem(_H-e_list,_G-john),
_F-arg_ne_list(_E-sem(_D-e_list,_C-mary),_B-e_list)),_A-call_up)),
Ws = [john,calls,up,mary] ? ;
```

```
no
```

The last two solutions in the example above are generated because the input string, `john calls mary up`, can be parsed both as a `sentence` type and as an `s` type.

7.8 Mini-interpreter (parsing only)

ALE contains a mini-interpreter that allows the user to traverse and edit an ALE parse tree. By default, the mini-interpreter is off when ALE is loaded. To turn the mini-interpreter on, simply type:

```
| ?- interp.

interpreter is active

yes
| ?-
```

To turn it off again, use `nointerp.` Any parse automatically stores the following information on the edges added to ALE's chart:

- Spanning nodes
- Substring spanned
- Creator
- Daughters (if any)

The spanning nodes are the nodes in the chart that the edge spans. The substring spanned is the concatenation of lexical items between the spanning nodes. If an edge was formed by the application of an ALE grammatical rule, its creator is that rule, with the daughters being the daughters of the rule, i.e., the `cat>`, and `cats>` of the rule). If the rule was created by EFD closure, the mini-interpreter will treat it as the user's rule it was created from, displaying the empty categories that had matched its daughters during EFD closure in the same way as daughters matched at run-time. If an edge represents an empty category, its creator is normally `empty`; but empty categories created during EFD closure will show the rules that created them, along with their empty category daughters. If an edge represents a lexical item, its creator is `lexicon`. In the case of empty categories not created by EFD closure and all lexical items, there are no daughters.

The status of the mini-interpreter has no effect on compilation. The same code is used regardless of whether the mini-interpreter is active or inactive. The mini-interpreter only has an effect on the run-time commands `rec/1,2,4,5` and `drec/1`.

When the mini-interpreter is active, `rec/1` operates in one of two modes: query-mode or go-mode. When the mini-interpreter is active, `rec/1` always begins in *query-mode*. In query-mode, the user is prompted just before any edge is added. Because ALE parses from right to left, the edges are encountered in that order. The prompt consists of a display of the feature structure for the edge, followed by the

mini-interpreter information for that edge, followed by an *action-line*, which lists the options available to the user. For example (from the HPSG grammar included in the ALE distribution):

```
| ?- rec([kim,sees,sandy]).

STRING:
0 kim 1 sees 2 sandy 3

word
QRETR list_quant
QSTORE e_set
SYNSEM synsem
       LOC loc
           CAT cat
               HEAD noun
                    CASE case
                    MOD none
                    PRD bool
               MARKING unmarked
               SUBCAT e_list
           CONT nom_obj
                INDEX [0] ref
                      GEN gend
                      NUM sing
                      PER third
                RESTR e_set

Edge created for category above:
      from: 2 to: 3
    string: sandy
      rule:  lexicon
 # of dtrs: 0

Action(add,noadd,go(-#),break,dtr-#,abort)?
|:
```

We see, in this example, the main action-line for `rec`. If the user selects `add`, the edge is added, and `rec` proceeds, in query-mode, as usual. If `noadd` is selected, the edge is not added, and `rec` proceeds in query-mode. In every ALE action-line, the first option is the one that ALE would have chosen if the mini-interpreter were disabled.

`go` puts the mini-interpreter into *go-mode*. In go-mode, `rec` proceeds as it would if the mini-interpreter were inactive, or to think of it another way, it functions as if the user always chose the first option on every action-line, but it does not stop to ask. As it adds the edges, it displays them, along with their mini-interpreter information. `go` suffixed with a number, e.g., `go-1`, puts the mini-interpreter into go-mode until it encounters an edge whose left node is that number, and then, beginning with that edge, automatically switches back into query-mode. With ALE's

current parsing strategy, `go-`*N* will remain in go-mode until it encounters the first edge corresponding to the (*N+1*)st lexical item in the string being parsed.

`break` simply invokes the Prolog `break` commmand, placing the user into a Prolog interpreter with a new break-level. Edges that have been added so far can be examined and retracted at this time. When the user pops out of the break, the current prompt is redisplayed.

`dtr-N` displays the `N`th daughter, its mini-interpreter information, and the action-line for `dtr`:

```
Action(retract,dtr-#,parent,abort)?
|:
```

`retract` removes the displayed edge (in this case, the daughter) from the chart. When the parse continues, ALE grammar rules will not be able to use that edge. The current edge (the parent of this daughter), however, can still be added. `dtr-N` has the same effect as in the `rec` action-line. `parent` returns to the current edge's parent and its action-line (either `rec` or `dtr`).

The mini-interpreter will not display any edge that has already been retracted. Note that if edges are retracted, there may be gaps in the sequence of chart-edge indices.

If `abort` is selected, the parse is aborted. All of the edges added so far remain in memory until the next `rec` statement. The edge that was displayed when `abort` was chosen is discarded.

Mini-interpreter information is also available through the run-time commands, `edge/2` and `edge/1`, e.g.:

```
| ?- edge(2,4).

COMPLETED CATEGORIES SPANNING: every toy

cat
QSTORE ne_list_quant
       HD every
          RESTR toy
                ARG1 [0] individual
          SCOPE proposition
          VAR [0]
       TL e_list
SYNSEM basic
       SEM [0]
       SYN np

Edge created for category above:
     index: 20
      from: 2 to: 4
    string: every toy
      rule:  np_det_nbar
 # of dtrs: 2
Action(retract,dtr-#,continue,abort)?
```

```
|:
```

Every edge that is actually asserted into the chart is assigned a unique number, called an *index*, which `edge/2` displays also. `retract` and `dtr` behave the same as in the `dtr` action-line. `continue` tells the mini-interpreter that the user is done traversing the parse tree rooted at the current edge, and to find more.

`edge/1` works exactly as `edge/2` does, except that its input is the unique index number that ALE stores with every edge.

7.9 Subsumption Checking (parsing only)

ALE can perform subsumption checking on edges during parsing. By default, it does not. To enable it, use the command:

```
| ?- subtest.

edge subsumption checking active


yes
| ?-
```

To turn it, off, use `nosubtest.`

When subsumption checking is enabled, ALE will only add a new feature structure to the chart between nodes n and m if there is no other edge currently spanning n and m whose feature structure subsumes the new one. If, instead, the new feature structure subsumes an existing one's, then the existing edge is retracted and replaced with the new one.

Note that if edges are retracted, there may be gaps in the sequence of chart-edge indices.

Extra compiled code is required in order to make subsumption checking more efficient. If subsumption checking is enabled when a grammar is compiled, this code will be compiled also. If it is disabled, the subsumption checking code will be compiled when the `subtest` command is given. Only in the former case, however, will subsumption checking be used during EFD closure to reduce the number of empty categories to consider at run-time. The empty categories and phrase structure rules must be recompiled (with `compile_rules`) if subsumption checking is enabled after the initial compilation of a grammar for empty categories to be tested for subsumption.

Our experience has been that subsumption checking is not required in most unification-based grammars, and should therefore be left disabled. It is useful only for those grammars which have true spurious ambiguity or redundancy. Grammars that incorporate some notion of thematic or functional structure for representing the meaning of a sentence normally realise structural ambiguities as semantic ambiguities that should be retained in the chart.

If both subsumption checking and the mini-interpreter are enabled, then the user may override either of these behaviours. In the former case, before the new feature structure is discarded, it will be displayed along with the discard action-line:

```
Action(noadd,continue,break,dtr-#,existing,abort)?
|:
```

`continue` instructs the parser to look for more subsuming edges. If no more are found, the new feature structure is added to the chart. `existing` displays the existing chart edge that subsumes the new one with the `edge/2` action-line. The rest behave as described above.

In the latter case, before an existing edge is retracted, its feature structure will be displayed along with the retract action-line:

```
Action(retract,continue,break,dtr-#,incoming,abort)?
|:
```

`retract` retracts the existing, subsumed edge and asserts the incoming feature structure. `incoming` displays the incoming feature structure, along with the `incoming` action-line:

```
Action(noadd,dtr-#,existing,abort)?
|:
```

the functions of whose options have already been described.

7.10 Source-Level Debugger

ALE also provides an XEmacs-based source-level debugger. This can only be used for parsing or definite clause resolution, and only with SICStus 3.8.6 or higher, and XEmacs 20.3 or higher. In future releases, a debugger with a more restricted functionality will be made available for users of SWI Prolog.

The ALE source-level debugger is implemented on top of the SICStus source-level debugger. The debugger provided with ALE has the complete functionality of the SICStus source-level debugger with ordinary Prolog programs; so you only need this one if you will need to debug both. SICStus debugger commands that are not explicitly mentioned in this section are not supported in ALE debugging.

ALE source code must occur in a single file in order to be debugged. To debug Prolog source code, please refer to the SICStus documentation. For both ALE and Prolog debugging, the prolog flag, `source_info`, needs to be turned on, using the command:

```
| ?- prolog_flag(source_info,_,on).
```

The SICStus XEmacs interface should do this automatically.

When a Prolog hook is encountered while debugging an ALE grammar, the SICStus debugger is automatically invoked. The hook will be embedded in a Prolog `call/1` statement. If the `leap` option of the SICStus debugger is used, the leap ends at the end of the hook — ALE will creep when it resumes control.

To install the ALE source-level debugger, follow the directions in the distribution file, `debugger/INSTALL`.

7.10.1 Running without XEmacs

To run the debugger without XEmacs, simply run SICStus Prolog from the directory with `debugger.pl` and type:

```
| ?- compile(debugger).
```

This assumes that `ale.pl` and the `debugger` subdirectory are located in the same directory. If ALE.PL has not been loaded already, this will compile `ale.pl` as well. There will be a warning message about buffer-mode when it loads, which can be ignored. Then type:

```
| ?- noemacs.
```

You can add a `noemacs/0` directive to the end of `debugger.pl` to do this automatically.

7.10.2 Running with XEmacs

To run the debugger with XEmacs, you must run SICStus Prolog and ALE within XEmacs as an inferior process. To do this:

1. set the EPROLOG environment variable to the command that runs SICStus Prolog in the shell that you will run XEmacs from (this is not necessary if the command is 'sicstus'),
2. run XEmacs from the directory with `debugger.pl`,
3. load the file to be debugged in Prolog major mode,
4. Use the XEmacs command, `M-x run-prolog`, to run SICStus prolog as an inferior process,
5. From the SICStus Prolog prompt, type:

   ```
   | ?- compile(debugger).
   ```

There is also a command:

```
| ?- emacs.
```

that will turn on the XEmacs interface, if it has been disabled by `noemacs/0`.

7.10.3 Debugger Commands

There are seven basic commands in the ALE debugger, four of which are variants of normal ALE commands. These four, `dcompile_gram/1`, `drec/1`, `dquery/1` and `dgen/1`, are the debugger variants of `compile_gram/1`, `rec/1`, `query/1` and `gen/1`, respectively. The other three, `dleash/1`, `dskip/1`, and `dclear_bps/0`, will be explained below.

Currently, the ALE source-level debugger can only debug grammars down to the level of feature structure unification, i.e., feature structure unification is treated as an atomic operation. Constraints and procedural attachments on types using `cons` and `goal` cannot be debugged either, nor can inequation enforcement, edge subsumption checking, or extensionalization. These will be possible in a future version. For now, `dcompile_gram/1` compiles a grammar in exactly the same way that ALE normally does to the point where it can be debugged. It then reopens the grammar file and uses the token-stream directly to index those parts of the grammar source code that it can debug by line number. `dcompile_gram/1` also requires that all of the ALE

source code for a grammar be in a single file, i.e., unlike ALE without the debugger, you cannot load other auxiliary files from within a grammar file. This restriction will also be lifted in a future version.

`dcompile_gram/1` also allows for tracing through the EFD closure algorithm, again down to the level of feature structure unification.

After a grammar has been compiled for debugging with `dcompile_gram/1`, `drec/1`, `dquery/1` and `dgen/1` can be used for parsing, definite clause resolution or generation, respectively, with the debugger. `dquery/1` works exactly as `query/1` does: it first finds most general satisfiers of the arguments and then searches for a solution with Prolog-style SLD-resolution using the clauses provided by the user. `drec/1` and `dgen/1` work exactly as `rec/1` and `gen/1` do, except that lexical rules are not compiled out, so that they can be debugged. Also remember that ALE parses right-to-left with EFD-closed rules and empty categories, and that it uses a semantic-head-driven generator. With parsing, the debugger can also be used in conjunction with the chart mini-interpreter described above for extra control of the chart. With generation, the debugger successively shows the construction of the pivot template, pivot matching, pivot checking and the linking of the pivot with the root. Generation of non-semantic-head daughters is performed recursively.

7.10.4 Debugger Ports and Steps

The ALE debugger is loosely based on the procedural box model of execution that many Prolog debuggers use. There are four kinds of ports, *call*, *exit*, *redo*, and *fail*. The ALE debugger does not support exception ports. A call port occurs at every initial invocation of a step that ALE takes in parsing or definite clause resolution, a list of which is given below. An exit port occurs at the successful completion of such a step; a redo port occurs when a subsequent step has failed and ALE backtracks into the current step to find more solutions, and a fail port occurs when a step has failed to produce any or any more solutions.

Consider, for example, what happens when ALE applies the following description to a feature structure that occurs in the lexical entry for the word, `foo`

```
foo --->(a
        ;f:b),
        g:c.
```

The first port we encounter is when ALE tries to add the type `a`. This is a call port:

```
Call: add type, a, to lex entry?
```

If this succeeds, then an exit port occurs:

```
Exit: add type, a, to lex entry?
```

and processing moves on to `g:c`. If this fails, then we must backtrack through adding `a` for more solutions (for example, if there is a disjunctive constraint on that type):

```
Redo: add type, a, to lex entry?
```

If there is another solution, then another exit port occurs. Otherwise, the next port is fail port:

```
Fail: add type, a, to lex entry?
```

and processing continues with the other disjunct, `f:b`. Certain steps in ALE, notably the depth-first rule application of the chart parser, are failure-driven loops. To indicate that these "failures" are actually a normal part of execution, they are displayed as, e.g.:

```
Finished:close chart edge under rule application?
```

Enforcing descriptions at a feature also counts as a step:

```
Call: enforce description on f value of lex entry?
```

If we agree to this, then the next step would be to add the type `b`

```
Call: add type, b, to value at f?
```

and so on.

The steps in a description whose ports the ALE debugger keeps track of are given in Figure 7.1: There are other kinds of steps besides description-level ones, that

Step	Kind	Example Message
Adding types	`desc`	`add type, a, to lex entry`
Adding `a_/1` atoms	`desc`	`add atom, foo(X), to lex entry`
Feature selection	`desc`	`enforce description on f value of lex entry`
Adding path equations	`desc`	`path equate [f,g] with [f,h]`
Adding inequations	`desc`	`inequate ...with description`
Unification (from shared variables)	`desc`	`unify ...with` Var
Macro substitution	`desc`	`substitute macro description for np/1`
Functional description evaluation	`desc`	`evaluate functional description, append/2`

Figure 7.1: Description-level Steps

pertain to ALE's built-in control for parsing and definite clause resolution. These steps, along with their *kind*, are given in the table in Figure 7.2. The steps for generation are given in Figure 7.3. Almost all of these involve sub-steps that enforce descriptions. Some, such as chart-edge closure, involve other sub-steps such as rule selection. Lexical rule application includes input and output morph application, as well as morph condition (given in a `when` clause) application. The one generation step of kind, `lr`, takes place when the pivot template is matched against a (base or derived) lexical entry. Unlike the compiled generator, the debugger does not compile lexical rules into the non-chain-rule selection step, so that the user can see their application. Instead, a base lexical entry is first selected without reference to the pivot template, then zero or more lexical rules are applied bottom up, until a derived entry is eventually unified with the pivot template. Lexical rules are, thus, treated as a special kind of unary chain rule. The length of these special chains is still controlled by `lex_rule_depth/1`, not `chain_length/1`.

Step	Kind	Example Message
Empty category derivation	`empty`	`empty category`
EFD Closure	`empty`	`close empty categories under rules`
EFD matching	`empty`	`apply daughter 1 of rule sentence1 to empty category`
Lexical entry derivation	`lex`	`derive seen from base entry: see`
Lexical rule application	`lr`	`apply lexical rule, passive, to see`
Input morph application	`lr`	`apply morph to input: see`
Output morph application	`lr`	`apply morph to input: see, output: seen`
Morph condition application	`lr`	`apply morph condition`
Functional description clause selection	`fun`	`evaluate functional clause for append/2`
Definite clause selection	`rel`	`evaluate relational` clause for `head_feature_principle/2`
Definite clause resolution	`rel`	`resolve goal, append/3`
Negated goal (`\+`) resolution	`rel`	`resolve negated goal`
Shallow cut (`->;`) execution	`rel`	`execute shallow cut`
Extensional identity (`@=`) check	`rel`	`resolve extensional identity`
Prolog hook call	`rel`	`resolve prolog hook: (num(N),write(N)`
Closing new chart edge under rules as leftmost daughter	`rule`	`close chart edge under rule application`
Rule selection	`rule`	`apply rule, schema1.`

Figure 7.2: Parsing and Definite Clause Resolution Steps

7.10.5 Leashing

With so many steps, and four possible ports, stepping through an entire parse in a large grammar would be a very trying experience. The ALE debugger provides three basic ways to filter through the steps to find points of interest in a parse or definite clause query.

The first is leashing. Leashing allows the user to distinguish at which steps information is simply displayed and at which steps the debugger stops and asks the user what to do. Unlike the SICStus debugger, leashing in the ALE debugger is a property of steps, not ports. The command to control leashing is `dleash/1`. The argument to dleash/1 consists of a sign, + or -, plus the kind of step. A + sign indicates that the debugger should stop and ask what to do at steps of that kind; and a - sign indicates that it should simply display the port and proceed. For example, to turn leashing off for empty categories, type:

```
| ?- dleash(-empty).

yes
```

There is also a special kind, `all`, that allows the user to turn on and off leashing on

Step	Kind	Example Message
Build semantic index	gen	build semantic index from root
Build pivot template	gen	build pivot template from index
Find pivot	gen	find pivot
Match pivot (lexical entry)	gen	match pivot against entry derived from see
Match pivot (empty category)	gen	match pivot against empty category
Match pivot (mother of non-chain rule)	gen	match pivot against mother of non-chain rule, sentence1
Apply chain rule	gen	apply chain rule, s, to, pivot
Pivot check	gen	check for link from pivot to root
Generate from pivot	gen	recursively generate from pivot
Generate pre-head daughters	gen	recursively generate pre-head daughters from pivot
Generate post-head daughters	gen	recursively generate post-head daughters from pivot
Connect pivot to root	gen	connect pivot to root
Connect chain node to root	gen	connect new chain node to root
Unify pivot template with lexical entry	lr	unify pivot template and see
Unify chain node	gen	unify mother of chain rule, s, with root

Figure 7.3: Generation Steps

all kinds of steps at once. The default leashing at start-up is `+all`.

If a kind of step is leashed, then the debugger will stop at every port for every step of that kind, and ask what to do. The possible responses are given in Figure 7.4: Not all responses are available at all ports. The kind of port (call, fail, etc.) is what determines the possible responses. The responses, `?` and `h` are always available, and list the other legal responses at the current port.

7.10.6 Skipping

Even leashing may not be enough for very large parses or queries because of the sheer number of ports displayed. The ALE debugger also provides a facility for *auto-skipping*. Whereas turning leashing off at a kind of step is like automatically answering `c` (advance to next port) at those steps, auto-skipping is like automatically answering `s`, which advances to the next exit or fail port of the current step without stopping at or even displaying the ports in between. The command for this is `dskip/1`, and its argument is of the same form as the argument to `dleash/1`. The signs have a different meaning, of course. For example, `dskip(+empty)` means that you want the debugger to auto-skip steps of kind `empty`, i.e., not stop and ask, whereas `dleash(+empty)` means that you want to leash steps of kind `empty`, i.e., stop and ask. When a step where auto-skipping is set is encountered, it is displayed with an automatic reply without stopping, e.g.:

```
Call: empty category? <auto-skip>
```

Input	Description	Ports
`?`	show available commands at current port	c,e,r,f
`h`	same as `?`	
`a`	abort processing	c,e,r,f
`f`	fail at this step (go to fail port)	c,e,r
`r`	retry this step (go from fail to call port)	f
`c`	advance to next port	c,e,r,f
LF	same as `c`	
`s`	advance to next exit/fail port of this step	c,r
`n`	advance to first port on new line of grammar file	c,e,r,f
`l`	advance to next breakpoint	c,e,r,f
`+`	set breakpoint at current line	c,e,r,f
`-`	clear breakpoint at current line	c,e,r,f
`@` *PrologGoal*	pass a goal to Prolog	c,e,r,f
`i`	toggle chart mini-interpreter	c,e,r,f
`d`	display current structure	c,e,r,f

Figure 7.4: Possible Responses at Debugger Ports

```
Exit: empty category? <auto-skipped>
Edge added: Number:0, Left:1, Right:1, Rule:empty
Call: close chart edge under rule application?
```

7.10.7 Breakpoints

The final kind of filtering is the breakpoint. In the ALE debugger, breakpoints are a property of lines in a grammar source file, not steps or ports. For a finer grain of resolution, it would be necessary to give each potential breakable step its own line in the input. By setting breakpoints and then using the `l` response, the debugger will advance to the next step whose line has a breakpoint without displaying any steps in between. If that step is not leashed or has auto-skipping set, the debugger acts accordingly after displaying it.

There are currently two ways to set a breakpoint. One is to use the `+` response from within the debugger at a step at whose line you wish to set a breakpoint. The other is only available when the debugger is used with an installation of XEmacs that supports XPM resources. In this case, when a source file is compiled a small glyph will be displayed at the left edge of every breakable line. Clicking on this glyph once with the left mouse button sets a breakpoint. Clicking again clears it. A breakpoint can also be cleared with the `-` response.

It is often the case that a line will have several breakable steps on it, for example, feature paths:

```
synsem:local:cat:head:verb,
qretr:e_list
```

If a breakpoint were set at the first line, then leaping from the call port for SYNSEM would still advance to the call port for LOCAL:

```
Call: enforce description on synsem value of lex entry? l
```

```
Call: enforce description on local value of value at synsem?
```

To avoid this, the response `n` is provided, for leaping automatically to the first port on a different line in the source file. The combination of `n` and `l` can be used to leap more effectively in files that pack many steps, particularly description steps, into one line.

All breakpoints can be cleared at once using the command, `dclear_bps/0`.

Chapter 8

ALE Keyword Summary

The following is a summary of keywords discussed in this manual, along with page references. A table of auxiliary keywords, those that only occur as arguments of other keyword operators, such as the `cat>` argument of a `rule`, will be provided in a future version.

A keyword of kind *Description* is one that occurs in an ALE description of a feature structure. One of kind *Def. Clause*, or *DCL*, is one that occurs in ALE's definite clause language. One of kind *Signature* is a declaration that occurs in an ALE signature. One of kind *Type* is an ALE type with special properties. One of kind ALE is a Prolog query (entered at the `| ?-` prompt) that can be used after ALE has been loaded (see p. 4). One of kind *Mini-interpreter* is a mini-interpreter command that appears in an interpreter action-line. One of kind *Debugger* is a Prolog query that can be used after the source-level debugger has been loaded. For debugger responses, the reader is referred to the table on page 100.

Keyword	Kind	Description	Page
`,`	Desc./DCL	Conjunction	22, 40
`-->`	Signature	Declare lexical entry.	42
`-> (;)`	Def. Clause	Shallow cut.	40
`:`	Description	Feature value.	22
`;`	Desc./DCL	Disjunction.	22, 40
`!`	Def. Clause	Cut.	40
`\+`	Def. Clause	Negation-by-failure.	40
`==`	Description	Path Equation.	22
`=@`	Def. Clause	Predefined token-identity definite clause predicate.	40
`=\=`	Description	Inequation.	22
`@`	Description	Macro instantiation.	28
`[...]`	Description	Predefined list macro.	29
`%`	Prolog	Comment delimiter.	7

Keyword	Kind	Description	Page
a_	Description/Signature	Built-in extensional atom.	20
abort	Mini-interpreter	Abort parse.	92
add	Mini-interpreter	Add the current edge.	91
approp	ALE	Show value restriction on a feature at a type.	73
assert	Prolog	Add clause to Prolog database.	40
bot	Type	In an ALE signature, this type must appear, and must subsume all of the other types.	6
break	Mini-interpreter	Invoke Prolog break.	92
chain_length	ALE	Set limit on chain rule sequence length.	62
compile_gram	ALE	Compile ALE signature (or parts of it — see table, p. 66).	66
compile	Prolog	Compile a Prolog file.	4
cons	Signature	Declare type constraint.	31, 41
consult	Prolog	Load Prolog file (such as an ALE signature) into database.	4
continue	Mini-interpreter	Proceed to look for more (subsuming/subsumed) edges.	93,94
control-c	Prolog	Prolog interrupt.	5
control-z	Unix	Unix interrupt.	5
dclear_bps	Debugger	Clear all breakpoints in current grammar file.	101
dcompile_gram	Debugger	Compile grammar file for source-level debugging.	95
dgen	Debugger	Generate with source-level debugger.	96
dleash	Debugger	Set or remove leashing on a kind of step.	98
dquery	Debugger	Evaluate a definite clause with source-level debugging.	96
drec	Debugger	Parse with source-level debugger.	96
dskip	Debugger	Set or remove auto-skipping on a kind of step.	99
dtr-*N*	Mini-interpreter	Display *N*th daughter edge of current edge.	92
edge	ALE	Show a chart edge.	84
emacs	Debugger	Turn on XEmacs interface to source-level debugger.	95
empty	ALE	Show empty categories.	80
empty	Signature	Declare empty category.	44
export_words	ALE	Send list of words in lexicon to stream	80
existing	Mini-interpreter	Display edge that subsumes new feature structure.	94
ext	Signature	Declare extensional types.	19
feature	ALE	Test if feature exists.	72
generate	ALE	Tell compiler to produce code for generation only.	66

Keyword	Kind	Description	Page
`gen`	ALE	Generate a string using the compiled generator.	87,89
`go`	Mini-interpreter	Add current and all subsequent edges.	91
`go-`*N*	Mini-interpreter	Add current and all subsequent edges until node *N* is reached.	91
`halt`	Prolog	Exit from Prolog.	5
`if`	Def. Clause	Definite clause language equivalent of `:-`.	38
`incoming`	Mini-interpreter	Display incoming edge that subsumes existing edge.	94
`interp`	ALE	Turn on mini-interpreter.	90
`intro`	Signature	Declare appropriate features for type.	13
`introduce`	ALE	Test if a feature was introduced by a type.	72
`iso_desc/2`	ALE	Test whether two descriptions evaluate to the same feature structure.	75
`lex`	ALE	Show lexical entry.	79
`lex_compile`	ALE	Compile lexicon intermediate code (SICStus only)	68
`lex_consult`	ALE	Dynamically consult lexicon intermediate code	68
`lex_rule`	ALE	Show lexical rule.	81
`lex_rule`	Signature	Declare lexical rule.	46
`lex_rule_depth`	ALE	Set bound on lexical rule application.	46
`list`	Type	This type, along with types `e_list` and `ne_list`, and features `HD` and `TL`, must be defined in an ALE signature in order to use the predefined `[...]` macro in descriptions, or the `cats>` list-argument operator in grammatical rules.	29, 54
`macro`	ALE	Show macro definition.	81
`macro`	Signature	Declare macro.	27
`mgsat`	ALE	Find most general satisfier(s) of a type.	73
`no_write_feat`	ALE	Hide a feature and its value.	76
`no_write_type`	ALE	Hide a type.	76
`noadd`	Mini-interpreter	Do not add the current edge.	91
`noemacs`	Debugger	Turn off XEmacs interface to source-level debugger.	95
`nointerp`	ALE	Turn off mini-interpreter.	90
`nosubtest`	ALE	Disable edge subsumption checking.	93
`parent`	Mini-interpreter	Return to parent edge.	92
`parse`	ALE	Tell compiler to produce code for parsing only.	66

Keyword	Kind	Description	Page
parse_and_gen	ALE	Tell compiler to produce code for parsing and generation.	66
prolog	Def. Clause	Definite clause hook to Prolog.	40
query	ALE	Evaluate a definite clause.	77
rec	ALE	Parse a string.	82,84–86
rec_best	ALE	Parse first parsable string in a list of strings	87
rec_list	ALE	Parse a list of strings	86,87
retract	Mini-interpreter	Retract currently displayed edge.	92
retract	Prolog	Remove clause from Prolog database.	40
retractall_lex	ALE	Retract all of a word's entries from lexicon	68
retract_lex	ALE	Retract a word's entry from lexicon	68
rule	ALE	Show grammatical rule.	80
rule	Signature	Declare grammatical rule.	50
semantics	ALE	Declares a semantics definite clause predicate.	57
show_clause	ALE	Show a definite clause.	76
show_cons	ALE	Show constraint for a type.	72
show_type	ALE	Show subtypes, supertypes, constraint and most general satisfiers for a type.	73
sub	Signature	Declare subtyping relationship.	6
subtest	ALE	Enable edge subsumption checking, and, if necessary, compile code for it.	93
sub_type	ALE	Test subsumption between two types.	72
true	Def. Clause	Definite clause that is always satified (also used to construct ground clauses in def. clause language).	38
type	ALE	Test if type exists.	71
unify_type	ALE	Unify two types.	72
update_lex	ALE	Add new entries to lexicon	68
write_feat	ALE	Don't hide a feature.	76
write_feats	ALE	Don't hide any features.	76
write_type	ALE	Don't hide a type.	76
write_types	ALE	Don't hide any types.	76

HDRUG: A Graphical User Environment for Natural Language Processing in Prolog

Hdrug is an environment to develop logic grammars / parsers / generators for natural languages. The package is written in Sicstus Prolog version 3 and uses library(tcltk) to implement its user interface. Tcl/Tk is a powerful script language to develop applications for the X-windows environment.

Hdrug offers various tools to visualize lexical entries, grammar rules, definite-clause definitions, parse trees, feature structures, lexical- rule- and type-hierarchies, graphs of the comparison of different parsers on a corpus of test sentences etc., in a Tk widget, LaTeX/DVI format, and the Clig system.

The package comes with a number of example grammars, including the grammars to be found in the distribution of the ALE system.

Hdrug allows for easy comparison of different parsers/generators; it has extensive possibilities to compile feature equations into Prolog terms; it can produce graphical (Tk), and ordinary Prolog output of trees, feature structures, Prolog terms (and combinations thereof), plotted graphs of statistical information, and tables of statistical information. Etc. Etc.

Using just menu's and buttons it is possible to parse sentences, generate sentences from logical form representations, view the parse trees that are derived by the parser or generator, change a particular version of the parser on the fly, compare the results of parsing the same sentence(s) with a set of different parsers, etc.

HDRUG was designed and implemented by Gertjan van Noord. The HDRUG home-page is `http://www.let.rug.nl/~vannoord/Hdrug/`.

Pleuk Grammar Development Environment

For those using SICStus 2.1#9 under X windows, the Pleuk grammar development shell has been adapted for ALE. Pleuk provides a graphical user interface, facilities for maintaining and testing corpora, and an interactive, incremental derivation checker. Pleuk is available free of charge from:

```
ftp.cogsci.ed.ac.uk:/pub/pleuk
```

The file README contains instructions for downloading the system. Pleuk has been ported to Sun SPARCs SunOS 4.* and HP-UX. For more information, send email to pleuk@cogsci.ed.ac.uk. Pleuk was developed by Jo Calder and Chris Brew of the Human Communication Research Centre at the University of Edinburgh, Kevin Humphreys of the Centre for Cognitive Science at the University of Edinburgh, and Mike Reape, of the Computer Science Department, Trinity College, Dublin.

As of this release, Pleuk will not work under SICStus 3.0 or later.

Chapter 9

References

This collection of references only scratches the surface of the relevant literature. A much more complete survey of the historical perspective on typed unification grammars and programs can be found in Carpenter (1992), and in subsequent papers in *ACL*, *EACL*, *COLING*, etc.

Aït-Kaci, H. (1991). The WAM: A (Real) Tutorial. MIT Press, Cambridge, Massachusetts.

The best available introduction to Prolog compiler technology, focusing on Warren's Abstract Machine for Prolog.

Aït-Kaci, H. (1986a). An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351.

Seminal work in sorted feature structures, based on Aït-Kaci's 1984 University of Pennsylvania dissertation. Focuses on general constraint resolution.

Aït-Kaci, H., and Nasr, R. (1986b). LOGIN: A logical programming language with built-in inheritance. *Journal of Logic Programming*, 3:187–215.

The first application of feature structures to logic programming. Includes sorted, but not typed feature structures. Also includes good details on the Martelli and Montanari (1984) unification algorithm applied to feature structures.

Aït-Kaci, H. (1984). A Lattice-Theoretic Approach to Computation based on a Calculus of Partially Ordered Type Structures. Univ. of Pennsylvania dissertation.

Aït-Kaci's introduction of sorted ψ-terms, which are like our feature structures only without the appropriateness conditions, inequations and extensionality. An appendix contains a coding of the Zebra Puzzle, a benchmark logic puzzle for constraint resolution.

Carpenter, B. (1992) *The Logic of Typed Feature Structures.* Cambridge Tracts in Theoretical Computer Science 32, Cambridge University Press, New York.

Contains all the theoretical details behind the ALE feature structures, description language and applications. A must for fully understanding ALE and a number of related variations.

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

A description of the theoretical underpinnings of ALE 2.0, including the data structures, type inference mechanism, description resolution, parsing, and inequation solving.

Colmerauer, A. (1987). Theoretical model of prolog II. In van Canegham, M., and Warren, D. H., editors, *Logic Programming and its Application*, 1–31. Ablex, Norwood, New Jersey.

Describes unification with cyclic terms and inequations in a logic programming environment.

Gazdar, G., and Mellish, C. S. (1989). *Natural Language Processing in Prolog.* Addison-Wesley, Reading, Massachusetts.

An introduction to computational linguistics using Prolog. Also contains a very general introduction to simple PATR-II phrase structure grammars, including simple implementations of unification and parsing algorithms. A version is also available using Lisp.

Höhfeld, M., and Smolka, G. (1988). Definite relations over constraint languages. LILOG–REPORT 53, IBM – Deutschland GmbH, Stuttgart.

Highly theoretical description of a constraint logic programming paradigm, including an application to feature structures similar to those used in LOGIN.

Jaffar, J. (1984). Efficient unification over infinite terms. *New Generation Computing*, 2:207–219.

Unification algorithm for possibly cyclic terms in Prolog II. Includes quasi-linear complexity analysis.

Kasper, R. T., and Rounds, W. C. (1990). The logic of unification in grammar. *Linguistics and Philosophy*, 13(1):35–58.

The details of Kasper and Rounds original feature structure description system and related theorems.

Martelli, A., and Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282.

The Union/Find unification algorithm used by ALE, which was adapted to the cyclic case by Jaffar (1984).

Mastroianni, M. (1993) Attribute-Logic Phonology. Carnegie Mellon University Laboratory for Computational Linguistics Technical Report CMU-LCL-93-4. Pittsburgh.

The description and motivation for an attribute-logic approach to phonology. Includes extensive discussion of its implementation in ALE, including syllable structure and morphologically conditioned effects such as epenthesis, harmony and assimilation.

O'Keefe, R. A. (1990) *The Craft of Prolog.* MIT Press, Cambridge, Massachussetts.

Best text on advanced programming techniques using Prolog compilers. Should read Sterling and Shapiro's introduction as a pre-requisite.

Penn, G. (1993). A Utility for Typed Feature Structure-based Grammatical Theories. Technical Report. Laboratory for Computational Linguistics, Carnegie Mellon University, Pittsburgh.

This project served as the basis of the version 2.0 updates of ALE. The report details these updates, including the algorithms used to implement them and other efficiency issues. It also describes Penn's implementation of Head-Driven Phrase Structure Grammar (HPSG), as represented in the first eight chapters of (Pollard and Sag 1994).

Penn, G. (1999). A Parsing Algorithm to Reduce Copying in Prolog. Arbeitspapier des Sonderforschungsbereichs 340, Nr. 137.

A presentation of the Empty-First-Daughter closure algorithm, which can be used to reduce copying in Prolog-based parsers.

Penn, G., and Carpenter, B. (1993). Three Sources of Disjunction in a Typed Feature Structure-based Resolution System. *Feature Formalisms and Linguistic Ambiguity*, H. Trost ed. Ellis Horwood, New York.

A presentation of the principal sources of complexity in solving constraint puzzles, such as the Zebra Puzzle, a simplified version of which is presented in this manual; and an outline of steps taken to cope with them in the reversible general constraint resolver which was the precursor to ALE.

Penn, G. and Popescu, O. (1997). Head-Driven Generation and Indexing in ALE. *Proceedings of Workshop on Computational Environments for Grammar Development and Linguistic Engineering (ENVGRAM)*, 35th ACL / 8th EACL.

Describes the implementation of ALE's head-driven generator, and a simple indexing strategy for lexical entries during generation.

Popescu, O. (1996). Head-Driven Generation for Typed Feature Structures. Carnegie Mellon University MS Project.

The extension of semantic-head-driven generation to typed feature structures used in ALE.

Pereira, F. C. N., and Shieber, S. M. (1987). *Prolog and Natural-Language Analysis*. Volume 10 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford.

Excellent introduction to the use of term unification grammars in natural language. Includes a survey of Prolog, parsing algorithms and many sample grammar applications in syntax and semantics.

Pollard, C. J. (in press). Sorts in unification-based grammar and what they mean. In Pinkal, M., and Gregor, B., editors, *Unification in Natural Language Analysis*. MIT Press, Cambridge, Massachusetts.

Contains the original extension of Rounds and Kasper's logical language to sorts. Also motivates the use of sorts in natural language grammars.

Pollard, C. J., and Sag, I. A. (1994). *Head-driven Phrase Structure Grammar*. Chicago University Press, Chicago.

The primary grammar formalism which motivated the construction of the ALE system. Provides many examples of how typed feature structures and their descriptions are employed in a sophisticated natural language application.

Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar*. Volume 4 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford.

Best source for getting acquainted with the application of feature structures and their descriptions to natural language grammars.

Shieber, S. M., Uszkoreit, H., Pereira, F. C. N., Robinson, J., and Tyson, M. (1983). The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*. Volume 1894 of *SRI Final Report*, SRI International, Menlo Park, California.

Original document describing the PATR-II formalism.

Shieber, S. M., Pereira, C. N., van Noord, G., Moore, R. C. (1990). Semantic-Head-Driven Generation. In *Computational Linguistics*, Vol. 16(1):30–42.

The main reference for a description of the semantic- head-driven generation algorithm.

Smolka, G. (1988a). A feature logic with subsorts. LILOG–REPORT 33, IBM – Deutschland GmbH, Stuttgart.

An alternative logic to that of Rounds and Kasper, which includes sorts, variables and general negation.

Smolka, G. (1988b). Logic programming with polymorphically order-sorted types. LILOG–REPORT 55, IBM – Deutschland GmbH, Stuttgart.

An application of ordered term unification to typed logic programming.

Sterling, L., and Shapiro, E. Y. (1986). *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, Massachusetts.

Best general introduction to logic programming in Prolog.

van Noord, G. (1989). *BUG: A Directed Bottom-Up Generator for Unification Based Formalisms*. Working Papers in Natural Language Processing, Katholieke Universiteit Leuven, Stichting Taaltechnologie Utrecht.

Proposes the first semantic-head-driven generation algorithm.

Appendix A

Sample Grammars

A.1 English Syllabification Grammar

```
% Signature
% =========

bot sub [unit,list,segment].
  unit sub [cluster,syllable,word]
       intro [first:segment,
              last:segment].
    cluster sub [consonant_cluster, vowel_cluster]
            intro [segments:list_segment].
      consonant_cluster sub [onset,coda].
        onset sub [].
        coda sub [].
      vowel_cluster sub [].
    syllable sub []
             intro [syllable:list_segment].
    word sub []
         intro [syllables:list_list_segment].
  segment sub [consonant,vowel].
    consonant sub [sibilant,obstruent,nasal,liquid,glide].
      sibilant sub [s,z].
        s sub [].
        z sub [].
      obstruent sub [p,t,k,b,d,g].
        p sub [].
        t sub [].
        k sub [].
        b sub [].
        d sub [].
        g sub [].
      nasal sub [n,m].
        n sub [].
        m sub [].
```

```
      liquid sub [l,r].
        l sub [].
        r sub [].
      glide sub [y,w].
        y sub [].
        w sub [].
    vowel sub [a,e,i,o,u].
      a sub [].
      e sub [].
      i sub [].
      o sub [].
      u sub [].
  list sub [e_list,ne_list,list_segment,list_list_segment].
    e_list sub [].
    ne_list sub [ne_list_segment,ne_list_list_segment]
            intro [hd:bot,
                   tl:list].
    list_segment sub [e_list,ne_list_segment].
      ne_list_segment sub []
                      intro [hd:segment,
                             tl:list_segment].
    list_list_segment sub [e_list,ne_list_list_segment].
      ne_list_list_segment sub []
                           intro [hd:list_segment,
                                  tl:list_list_segment].


% Rules
% =====

word_schema_rec rule
(word,
 syllables:[Syllable|Syllables],
 first:First1,
 last:Last2)
===>
cat> (syllable,
      syllable:Syllable,
      first:First1,
      last:Last1),
cat> (word,
      syllables:Syllables,
      first:First2,
      last:Last2),
goal> (\+ less_sonorous(Last1,First2)).

word_schema_base rule
(word,
```

```
 syllables:[Syllable],
 first:First,
 last:Last)
===>
cat> (syllable,
     syllable:Syllable,
     first:First,
     last:Last).

v_syllable rule
(syllable,
 syllable:[Vowel],
 first:Vowel,
 last:Vowel)
===>
cat> (vowel,Vowel).

vc_syllable rule
(syllable,
 syllable:[Vowel|Segs1],
 first:Vowel,
 last:Last)
===>
cat> (vowel,Vowel),
cat> (coda,
     segments:Segs1,
     last:Last).

cv_syllable rule
(syllable,
 syllable:Segs,
 first:First,
 last:Vowel)
===>
cat> (onset,
     segments:Segs1,
     first:First),
cat> (vowel,Vowel),
goal> append(Segs1,[Vowel],Segs).

cvc_syllable rule
(syllable,
 syllable:Segs,
 first:First,
 last:Last)
===>
cat> (onset,
     segments:Segs1,
```

```
     first:First),
cat> (vowel,Vowel),
cat> (coda,
     segments:Segs2,
     last:Last),
goal> append(Segs1,[Vowel|Segs2],Segs).

consonant_cluster_base rule
(consonant_cluster,
 segments:[Consonant],
 first:Consonant,
 last:Consonant)
===>
cat> (consonant,Consonant).

onset rule
(onset,
 segments:[Consonant1|Consonants],
 first:Consonant1,
 last:Consonant3)
===>
cat> (consonant,Consonant1),
cat> (onset,
     segments:Consonants,
     first:Consonant2,
     last:Consonant3),
goal> less_sonorous(Consonant1,Consonant2).

coda rule
(coda,
 segments:[Consonant1|Consonants],
 first:Consonant1,
 last:Consonant3)
===>
cat> (consonant,Consonant1),
cat> (coda,
     segments:Consonants,
     first:Consonant2,
     last:Consonant3),
goal> less_sonorous(Consonant2,Consonant1).


% Lexicon
% =======

p ---> p.
t ---> t.
k ---> k.
```

```
b ---> b.
d ---> d.
g ---> g.
s ---> s.
z ---> z.
n ---> n.
m ---> m.
l ---> l.
r ---> r.
y ---> y.
w ---> w.
a ---> a.
e ---> e.
i ---> i.
o ---> o.
u ---> u.


% Definite Clauses
% ================

less_sonorous_basic(sibilant,obstruent) if true.
less_sonorous_basic(obstruent,nasal) if true.
less_sonorous_basic(nasal,liquid) if true.
less_sonorous_basic(liquid,glide) if true.
less_sonorous_basic(glide,vowel) if true.

less_sonorous(L1,L2) if
  less_sonorous_basic(L1,L2).
less_sonorous(L1,L2) if
  less_sonorous_basic(L1,L3),
  less_sonorous(L3,L2).

append([],Xs,Xs) if true.
append([X|Xs],Ys,[X|Zs]) if
  append(Xs,Ys,Zs).
```

A.2 Categorial Grammar with Cooper Storage

```
% Signature
% =========

bot sub [cat,synsem,syn,sem_obj,list_quant].
  cat sub []
      intro [synsem:synsem,
             qstore:list_quant].
  synsem sub [functional, basic].
    functional sub [forward,backward]
               intro [arg:synsem,
                      res:synsem].
      forward sub [].
      backward sub [].
    basic sub []
          intro [syn:syn, sem:sem_obj].
  syn sub [np,s,n].
    np sub [].
    s sub [].
    n sub [].
  sem_obj sub [individual, proposition, property].
    individual sub [j,m].
      j sub [].
      m sub [].
    property sub []
             intro [ind:individual,
                    body:proposition].
    proposition sub [logical,quant,run,hit,nominal].
      logical sub [and,or].
        and sub []
            intro [conj1:proposition,
                   conj2:proposition].
        or sub []
           intro [disj1:proposition,
                  disj2:proposition].
      quant sub [every,some]
            intro [var:individual,
                   restr:proposition,
                   scope:proposition].
        every sub [].
        some sub [].
      run sub []
          intro [runner:individual].
      hit sub []
          intro [hitter:individual,
                 hittee:individual].
      nominal sub [kid,toy,big,red]
```

```
              intro [arg1:individual].
        kid sub [].
        toy sub [].
        big sub [].
        red sub [].
  list_quant sub [e_list, ne_list_quant].
    e_list sub [].
    ne_list_quant sub []
                  intro [hd:quant,
                         tl:list_quant].



% Lexicon
% =======

kid --->
  @ cn(kid).

toy --->
  @ cn(toy).

big --->
  @ adj(big).

red --->
  @ adj(red).

every --->
  @ gdet(every).

some --->
  @ gdet(some).

john --->
  @ pn(j).

runs --->
  @ iv((run,runner:Ind),Ind).

hits --->
  @ tv(hit).


% Grammar
% =======

forward_application rule
```

```
(synsem:Z,
 qstore:Qs)
===>
cat> (synsem:(forward,
      arg:Y,
      res:Z),
      qstore:Qs1),
cat> (synsem:Y,
      qstore:Qs2),
goal> append(Qs1,Qs2,Qs).


backward_application rule
(synsem:Z,
 qstore:Qs)
===>
cat> (synsem:Y,
      qstore:Qs1),
cat> (synsem:(backward,
      arg:Y,
      res:Z),
      qstore:Qs2),
goal> append(Qs1,Qs2,Qs).


s_quantifier rule
(synsem:(syn:s,
         sem:(Q,
              scope:Phi)),
 qstore:QsRest)
===>
cat> (synsem:(syn:s,
              sem:Phi),
      qstore:Qs),
goal> select(Qs,Q,QsRest).



% Macros
% ======

cn(Pred) macro
  synsem:(syn:n,
          sem:(body:(Pred,
                     arg1:X),
               ind:X)),
  @ quantifier_free.
```

```
gdet(Quant) macro
  synsem:(forward,
          arg: @ n(Restr,Ind),
          res: @ np(Ind)),
  qstore:[@ quant(Quant,Ind,Restr)].

quant(Quant,Ind,Restr) macro
  (Quant,
   var:Ind,
   restr:Restr).

adj(Rel) macro
  synsem:(forward,
          arg: @ n(Restr,Ind),
          res: @ n((and,
                    conj1:Restr,
                    conj2:(Rel,
                           arg1:Ind)),
                    Ind)),
  @ quantifier_free.

n(Restr,Ind) macro
  syn:n,
  sem:(body:Restr,
       ind:Ind).

np(Ind) macro
  syn:np,
  sem:Ind.

pn(Name) macro
  synsem: @ np(Name),
  @ quantifier_free.

iv(Sem,Arg) macro
  synsem:(backward,
          arg: @ np(Arg),
          res:(syn:s,
               sem:Sem)),
  @ quantifier_free.

tv(Rel) macro
  synsem:(forward,
          arg:(syn:np,
               sem:Y),
          res:(backward,
               arg:(syn:np,
                    sem:X),
```

```
               res:(syn:s,
                    sem:(Rel,
                         hitter:X,
                         hittee:Y)))),
  @ quantifier_free.

quantifier_free macro
  qstore:[].


% Definite Clauses
% ================

append([],Xs,Xs) if
  true.
append([X|Xs],Ys,[X|Zs]) if
  append(Xs,Ys,Zs).


select([Q|Qs],Q,Qs) if
  true.
select([Q1|Qs1],Q,[Q1|Qs2]) if
  select(Qs1,Q,Qs2).
```

A.3 Simple Generation Grammar

```
% An implementation in {\sc ale} of the grammar in Shieber & al,
% "Semantic-Head-Driven Generation", CL 16-1, 1990.

% Signature
% =========

bot sub [pred, list, sem, form, agr, sign].
  pred sub [decl, imp, love, call_up, leave, see, john, mary, mark,
            friends, often, friend, up, you, i].
    decl sub []. imp sub [].
    leave sub []. love sub []. call_up sub []. see sub [].
    john sub []. mary sub []. mark sub [].
    friends sub []. friend sub [].
    often sub []. up sub [].
    you sub []. i sub [].
  list sub [e_list, ne_list, arg_list, subcat_list].
    e_list sub [].
    ne_list sub [arg_ne_list, subcat_ne_list]
            intro [hd:bot, tl:list].
    arg_list sub [e_list, arg_ne_list].
      arg_ne_list sub [] intro [hd:sem, tl:arg_list].
    subcat_list sub [e_list, subcat_ne_list].
      subcat_ne_list sub [] intro [hd:sign, tl:subcat_list].
  sem sub [] intro [pred:pred, args:arg_list].
  form sub [finite, nonfinite].
    finite sub [].
    nonfinite sub [].
  agr sub [sg1, sg2, sg3, pl1, pl2, pl3].
    sg1 sub []. sg2 sub []. sg3 sub [].
    pl1 sub []. pl2 sub []. pl3 sub [].
  sign sub [sentence, verbal, np, adv, p]
       intro [sem:sem].
    sentence sub [].
    verbal sub [s, vp] intro [form:form].
      s sub [].
      vp sub [] intro [subcat:subcat_list].
    np sub [det, n]
       intro [agr:agr, arg:sem].
      det sub [] intro [np_sem:sem].
      n sub [].
    adv sub [] intro [varg:sem].
    p sub [].
ext([sg1,sg2,sg3,pl1,pl2,pl3]).

% Lexicon
% =======
```

```
love --->
  vp, form:nonfinite,
  subcat:[(np,sem:Obj),(np,sem:Subj)],
  sem:(pred:love,args:[Subj,Obj]).

call --->
  vp, form:nonfinite,
  subcat:[(np,sem:Obj),(p,sem:(pred:up,args:[])),(np,sem:Subj)],
  sem:(pred:call_up,args:[Subj,Obj]).

call --->
  vp, form:nonfinite,
  subcat:[(p,sem:(pred:up,args:[])),(np,sem:Obj),(np,sem:Subj)],
  sem:(pred:call_up,args:[Subj,Obj]).

leave --->
  vp, form:nonfinite,
  subcat:[(np,sem:Subj)],
  sem:(pred:leave,args:[Subj]).

see --->
  vp, form:nonfinite,
  subcat:[(np,sem:Obj),(np,sem:Subj)],
  sem:(pred:see,args:[Subj,Obj]).

see --->
  vp, form:nonfinite,
  subcat:[(s,form:finite,sem:Obj),(np,sem:Subj)],
  sem:(pred:see,args:[Subj,Obj]).

john --->
  np, agr:sg3, sem:(pred:john,args:[]).

mary --->
  np, agr:sg3, sem:(pred:mary,args:[]).

mark --->
  np, agr:sg3, sem:(pred:mark,args:[]).

friends --->
  np, agr:pl3, sem:(pred:friends,args:[]).

friend --->
  n, agr:sg3, arg:X, sem:(pred:friend,args:[X]).

i --->
  np, agr:sg1, sem:(pred:i,args:[]).
```

```
you --->
  np, agr:sg2, sem:(pred:you,args:[]).

often --->
  adv, varg:VP, sem:(pred:often,args:[VP]).

up --->
  p, sem:(pred:up,args:[]).


% Lexical Rules
% =============

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).

non_sg3 lex_rule (vp, form:nonfinite, subcat:Subcat, sem:Sem) **>
                 (vp, form:finite, subcat:NewSubcat, sem:Sem)
  if add_nonsg3(Subcat,NewSubcat)
  morphs X becomes X.


% Grammar Rules
% =============

sentence1 rule
  (sentence,sem:(pred:decl,args:[S])) ===>
  cat> (s,form:finite,sem:S).

sentence2 rule
  (sentence,sem:(pred:imp,args:[S])) ===>
  cat> (vp,form:nonfinite,
        subcat:[(np,sem:(pred:you,args:[]))],sem:S).

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

vp1 rule
  (vp,form:Form,subcat:Subcat,sem:S) ===>
  sem_head> (vp,form:Form,subcat:[Compl|Subcat],sem:S),
  cat> Compl.
```

```
vp2 rule
  (vp,form:Form,subcat:[Subj],sem:S) ===>
  cat> (vp,form:Form,subcat:[Subj],sem:VP),
  sem_head> (adv,varg:VP,sem:S).


% Semantics Directive
% ====================

semantics sem1.


% Definite Clauses
% ================

sem1(sem:S,S) if true.

add_sg3([(np,sem:Sem)],[(np,agr:sg3,sem:Sem)]) if !, true.
add_sg3([Cat|Cats],[Cat|NewCats]) if add_sg3(Cats,NewCats).

add_nonsg3([(np,sem:Sem)],[(np,agr:(=\=sg3),sem:Sem)]) if !, true.
add_nonsg3([Cat|Cats],[Cat|NewCats]) if add_nonsg3(Cats,NewCats).
```

Appendix B

Error and Warning Messages

B.1 Error Messages

`a_/1 atom declared subsumed by type` T

Subsumption over `a_/1` atoms has a fixed definition. Subtyping specifications with `a_/1` atoms are not allowed.

`add_to could not unify` FS_1 `and` FS_2

The unification between feature structures FS_1 and FS_2 failed.

`add_to could not unify paths` π `and` ϕ `in` FS

The unification between paths π and ϕ failed in feature structure FS failed.

`add_to could not inequate` FS_1 `and` FS_2

The inequation between features structures FS_1 and FS_2 could not been satisfied.

`add_to could not add feature` F `to` FS

The value of feature F is not unifiable with the similar value in feature structure FS.

`add_to could not add undefined macro` M `to` FS

Macro M used in feature structure FS is undefined.

`add_to could not add incompatible type` T `to` FS

Type T is not compatible with the type of feature structure FS.

`add_to could not add undefined type` T `to` FS

Type T in feature structure FS is undefined.

`add_to could not add ill formed complex description` D `to` FS

Description D is ill formed and could not be unified with feature structure FS

`appropriateness cycle following path` π `from type` T

There is a sequence of features π which must be defined for objects of type T where the value must be of type T.

`bot has appropriate features`

The most general type, $\bot$, cannot have any appropriate features.

`bot has constraints`

The most general type, $\bot$, must not have `cons` constraints.

`cats> value with sort` S `is not a valid list argument`

An argument of `cats>` was detected at run-time, which is not of a type subsumed by `list`.

`consistent` T_1 `and` T_2 `have multiple mgus` Ts

Types T_1 and T_2 have the non singleton set Ts as their set of most general unifiers.

`constraint declaration given for atom`

`a_/1` atoms must not have `cons` constraints.

`description uses unintroduced feature` F

A description uses the feature F which has not been defined as appropriate for any types.

`edge/2: arguments must be non-negative`

The arguments to edge/2 represent nodes in a parsing chart, and thus must be non-negative integers.

`edge/2: first argument must be < second argument`

The arguments to edge/2 represent nodes in a parsing chart. Edges only span from one node to an equal or greater valued node. `edge/2` shows edges where the other node has a greater value. `empty/0` shows edges where the node has an equal value.

`extensional type` E `is not maximal`

Type E is declared extensional but does not observe the maximality restriction.

`feature` F `multiply introduced at` Ts

The feature F is introduced at the types in Ts, which are not comparable with one another.

`illegal variable occurence in` T `sub` Ss `(intro` FRs`)`

In subtype/feature specifications, neither T or any of the types in Ss or FRs, or any of the features in FRs can be unbound variables. If a value restriction is an `a_/1` atom, that atom can be unbound, or contain unbound variables, but the `a_/1` operator must still appear.

`incompatible restrictions on feature` F `at type` T `are` Ts

The inherited restrictions, consisting of types Ts, on the value of F at type T are not consistent.

`invalid line` ϕ `in rule`

A line of a grammar rule is neither a goal nor a category description.

`lexical rule` LR `lacks morphs specification`

The obligatory `morphs` part of lexical rule LR is missing.

`multiple constraint declaration error for` T

More than one `cons` declaration exists for type T.

`multiple feature specifications for type` T

The appropriate features of T can be introduced along with subtyping or by themselves, but there can only be one declaration of appropriate features.

`multiple specification for` F `in declaration of` T

More than one restriction on the value of feature F is given in the definition of type T.

`no lexical entry for` W

Expression W is used, but has no lexical entry.

`pathval: illegal path specified -` π

Path π is not a valid path specification for the given feature structure.

`rule` R `has multiple sem_head> specifications`

More than one semantic head declaration was found in grammar rule R.

`rule` R `has no cat> cats> or sem_head> specification`

The grammar rule named R is empty in that it does not have any daughter specification.

`rule` R `has wrongly placed sem_goal> specifications`

A `sem_goal>` specification occurs somewhere other than immediately before or immediately after a `sem_head>` specification in rule R.

`subtype/feature specification given for a_/1 atom`

Subsumption over `a_/1` atoms has a fixed definition, and they can have no features. Subtype or feature specifications for them are not allowed.

`subtyping cycle at` T

The subsumption relation specified is not anti-symmetric. It can be inferred that the type T is a proper subtype of itself.

`subtype` T_1 `used in` T_2 `undeclared`

Undefined type T_1 declared as subtype in definition of T_2.

T `multiply defined`

There is more than one definition of type T.

T `subsumes bot`

T is declared as subsuming the most general type, $\bot$.

T_1 `used in appropriateness definition of` T_2 `undeclared`

Undefined type T_1 used as value restriction in definition of T_2.

`undefined macro` M `used in description`

A description uses a macro which is not defined.

`undefined type` T `used in description`

A description uses a type T which is not defined.

`undefined feature` F `used in path` π

A path π of features uses undefined feature F in a description.

`unsatisfiable lexical entry for` W

Word W has a lexical entry which has no satisfying feature structure.

`upward closure fails for` F `in` $S1$ `and` $S2$

$S1$ subsumes $S2$, but the value restriction for F at $S1$ does not subsume the value restriction for F at $S2$.

B.2 Warning Messages

`=@ accessible by procedural attachment calls from constraint for` T

The built-in `=@` predicate (extensional identity check) is non-monotonic, so its use should be avoided in constraints attached to types.

`atom a_/1` *Atom* `is ground in declaration of` T

One of the appropriate features of T has a value restriction that is a ground `a_/1` atom. Every feature structure of type T will have the same value at this feature.

`homomorphism condition fails for` F `in` T_1 `and` T_2

It is not the case that the appropriateness restriction on the type $T = T_1 + T_2$ is the unification of the appropriateness restrictions on T_1 and T_2.

`lexical description for` W `is unsatisfiable`

Incompatibilities in the lexical description for word W could not produce a satisfying feature structure.

`no chain rules found`

All the grammar rules in the program were non-chain rules (no semantic heads).

`no definite clauses found`

There were no definite clause rules specified in the program.

`no features introduced`

There are no appropriate features for any types.

`no functional descriptions found`

There are no functional description definitions in the program.

`no lexical rules found`

There were no lexical rules specified in the program.

`no lexicon found`

There were no lexical entries specified in the program.

`no non_chain rules found`

All the grammar rules in the program were chain rules (with semantic heads).

`no` P `definite clause found`

A definition for definite clause predicate P, which was declared as the semantics predicate, was not found in the program.

`no phrase structure rules found`

There were no phrase structure rules specified in the program.

`no semantics specification found`

There was no specification of the semantics definite clause predicate in the program.

`no types defined`

There were no `sub` or `intro` declarations found in the program.

`unary branch from` T_1 `to` T_2

The only subtype of T_1 is T_2. In this situation, it is usually more efficient to elimate T_1 if every instance of T_1 is a T_2.

Appendix C

BNF for ALE

```
<desc> ::= <type>
         | <variable>
         | (<feature>:<desc>)
         | (<desc>,<desc>)
         | (<desc>;<desc>)
         | @ <macro_spec>
         | <func_spec>
         | a_ <prolog_term>
         | <path> == <path>
         | =\= <desc>

<type> ::= <prolog_functor>

<feature> ::= <prolog_atom>

<path> ::= list(<feature>)

<macro_def> ::= <macro_head> macro <desc>.

<macro_head> ::= <macro_name>
               | <macro_name>(<seq(var)>)

<macro_spec> ::= <macro_name>
               | <macro_name>(<seq(desc)>)

<func_def> ::= <func_spec> +++> <desc>.

<func_spec> ::= <func_name>
               | <func_name>(<seq(desc)>)

<clause> ::= <literal> if <goal>.

<literal> ::=  <pred_sym>
            |  <pred_sym>(<seq(desc)>)
```

```
<cut_free_goal> ::= true
                  | <literal>
                  | prolog(<prolog_goal>)
                  | (<cut_free_goal>,<cut_free_goal>)
                  | (<cut_free_goal>;<cut_free_goal>)
                  | (<desc> =@ <desc>)
                  | (<cut_free_goal> -> <cut_free_goal>)
                  | (<cut_free_goal> -> <cut_free_goal>
                                      ; <cut_free_goal>)
                  | (\+ <cut_free_goal>)

<goal> ::= true
         | <literal>
         | prolog(<prolog_goal>)
         | (<goal>,<goal>)
         | (<goal>;<goal>)
         | (<desc> =@ <desc>)
         | (<cut_free_goal> -> <goal>)
         | (<cut_free_goal> -> <goal> ; <goal>)
         | !
         | (\+ <goal>)

<lex_entry> ::=  <word> ---> <desc>.

<rule> ::=  <rule_name> rule <desc> ===> <rule_body>.

<rule_body> ::=  <sem_less_rule_body>
              |  <sem_less_rule_body>,<sem_rule_body>,
                 <sem_less_rule_body>

<sem_less_rule_body> ::= <rule_clause>
                       | <rule_clause>, <sem_less_rule_body>

<rule_clause> ::=  cat> <desc>
                |  cats> <desc>
                |  goal> <goal>

<sem_rule_body> ::= sem_head> <desc>
                  | sem_goal> <goal>, sem_head> <desc>
                  | sem_head> <desc>, sem_goal> <goal>
                  | sem_goal> <goal>, sem_head> <desc>,
                    sem_goal> <goal>

<lex_rule> ::= <lex_rule_name> lex_rule <lex_rewrite>
                               morphs <morphs>.
```

```
<lex_rewrite> ::= <desc> **> <desc>
                | <desc> **> <desc> if <goal>

<morphs> ::= <morph>
           | <morph>, <morphs>

<morph> ::= (<string_pattern>) becomes (<string_pattern>)
          | (<string_pattern>) becomes (<string_pattern>)
                               when <prolog_goal>

<string_pattern> ::= <atomic_string_pattern>
                   | <atomic_string_pattern>, <string_pattern>

<atomic_string_pattern> ::=  <atom>
                           |  <var>
                           |  <list(<var_char>)>
<var_char> ::= <char>
             | <var>

<seq(X)> ::= <X>
           | <X>, <seq(X)>

<empty_prod> ::= empty <desc>.

<type_spec> ::= <type> sub <list(<type>)>
              | <type> sub <list(<type>)>
                       intro <list(<frestr_spec>)>
              | <type> intro <list(<frestr_spec>)>

<frestr_spec> ::= <feature>:<type>
                | <feature>: a_ <prolog_term>

<cons_spec> ::= <type> cons <desc>
              | <type> cons <desc>
                       goal <goal>

<ext_spec> ::= ext(list(<type>))

<prog> ::= <prog_line>
         | <prog_line> <prog>
```

```
<prog_line> ::= <type_spec>
              | <ext_spec>
              | <cons_spec>
              | <macro_def>
              | <empty_prod>
              | <clause>
              | <rule>
              | <lex_entry>
              | <lex_rule>
```

Appendix D

Reference Card

```
Grammar
-------------------------------------------------------------------------

Type sub Subtypes (intro [F1:R1,...,Fn:Rn]).  (Subtyping)
Type intro [F1:R1,...,Fn:Rn].                 (Appropriateness)
ext([Type1,...,Typen]).                       (Extensionality)

(Types not otherwise declared, but used in the above definitions are
 assumed to be maximal and/or immediately subsumed by bot)

Type cons Desc                                (Type Constraint)
    (goal Goal).

Word ---> Desc.                               (Lexical Entry)
empty Desc.                                   (Empty Category)
RuleName lex_rule DescIn **> DescOut          (Lexical Rule)
                  (if Goal)
         morph X becomes [X,e,s]
                 when PrologGoal.
RuleName rule Desc ===>                       (Phrase Structure Rule)
                       cat> Desc/             [Daughter]
                       cats> ListDesc/        [List of Daughters]
                       sem_head> Desc/        [Semantic Head]
                       goal> Goal/            [Procedural Attachment]
                       sem_goal> Goal.        [P.A. to Semantic Head]

f(Desc1,...,Descn) if g(Desc1,...,Descn)/     (Definite Clause)
                      Desc1 =@ Desc2/         (Extensionality Check)
                      prolog(PrologGoal).     (Prolog Hook)

fun(Desc1,...,Descn) +++> DescResult.         (Function Declaration)
macro(X1,...,Xn) macro Desc.                  (Macro Declaration)
semantics Pred.                               (Semantic Pred. Declaration)
```

```
Compile-time Options
--------------------------------------------------------------------
| ?- parse/generate/parse_and_gen.        (Compilation Mode)
| ?- lex_consult/lex_compile.             (Lexicon Compilation Mode)
| ?- (no)adderrs.                         (Toggle Description Errors)
| ?- (no)subtest.                         (Toggle Subsumption Check)
| ?- chain_length(Num).                   (Chain Rule Length Bound)
| ?- lex_rule_depth(Num).                 (Lexical Rule Depth Bound)

Compilation
--------------------------------------------------------------------
| ?- (d)compile_gram(GramFile).           (Grammar Compilation)
| ?- update_lex(File).                    (Incremental Lexicon Update)
| ?- retract(all)_lex.                (Incremental Lexicon Retraction)

Grammar Inspection
--------------------------------------------------------------------
| ?- (no_)write_type(s).                  (Type Hiding/Showing)
| ?- (no_)write_feat(s).                  (Feature Hiding/Showing)
| ?- show_type Type.                      (Signature Inspection)
| ?- unify_type(Type1,Type2,LUB).         (Type Unification)
| ?- approp(Feat,Type,Restr).             (Appropriateness Inspection)
| ?- introduce(Feat,Type).                (Feature Introduction)
| ?- iso_desc(Desc1,Desc2).               (Extensional Identity)
| ?- mgsat Desc.                          (Most General Satisfier)
| ?- show_clause PredName(/Arity).        (Definite Clause)
| ?- lex Word.                            (Lexical Entry)
| ?- rule Rulename.                       (Phrase Structure Rule)
| ?- empty.                               (Empty Category)
| ?- macro Macro.                         (Macro Definition)
| ?- lex_rule Lexrulename.                (Lexical Rule)
| ?- export_words(Stream,Delim).          (Lexicon Export)

Execution
--------------------------------------------------------------------
| ?- (d)rec WordList.                     (Bottom-up Parsing)
| ?- (d)query Query.                      (SLD Resolution)
| ?- (d)gen Desc.                         (Head-Driven Generation)

Run-time Options
--------------------------------------------------------------------
| ?- (no)interp.                          (Toggle Mini-Interpreter)
| ?- edge(Left,Right).                    (Chart Edge Inspection)
| ?- dleash(+/-Kind).                     (Port Leashing)
| ?- dskip(+/-Kind).                      (Port Auto-Skipping)
| ?- dclear_bps.                          (Breakpoint Clearing)
```