ALE
The Attribute Logic Engine
User's Guide

Version *j3*

by

Bob Carpenter

December 1992

Report CMU-LCL-92-1

Laboratory for Computational Linguistics

139 Baker Hall
Department of Philosophy
Carnegie Mellon University
Pittsburgh, PA 15213

ALE

The Attribute Logic Engine

User's Guide

Version β

December 1992

Bob Carpenter
Laboratory for Computational Linguistics
Philosophy Deparmtent
Carnegie Mellon University
Pittsburgh, PA 15213

Preface

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 *0.*"

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 axe 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. The next version of this system, which should be available by Summer 1993, will be greatly improved 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.

Contents

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 axe 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 Rasper's attribute-value logic with variables. The definite clause programs allow disjunction, negation and cut, specified with Prolog syntax. Phrase structure grammars axe 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 definite clause programs, runs at rougly 1000 logical inferences per second (1000 LI/s) on a DECStation 5100. This is roughly 15% of the speed of the SICStus 2.1 interpreter, and about 1.5% as fast as the SICStus compiler running naive reverse on a 30-element list. The definite clause compiler performs last call optimization, but does not index arguments.

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.

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, bJ.7, 123, 'JohnDoe', '65$', and '_65a.' are constants, but A19, JohnDoe, B_112, _au8, and [dd,e] are not. 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 lXa, aXX, and JCy[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

[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.

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.

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.

Running Prolog

To fire up Prolog locally, you should contact your systems administrator. You should have either SICStus or Quintus 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.

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.

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

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, a 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.

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 1-6 and 10, which is necessary to use ALE.

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.

The relation of subtyping is only specified one step at a time, but is taken to be 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`. Removing the declaration of `bot` would violate this condition, as would adding an additional specification, such as simply adding `j sub [k,l]`, as `j` would not be a subtype of `bot`, or a declaration `m sub [bot]`, as `bot` would no longer be the most general type.

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).

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
QSTDRE e_list
SYNSEM basic
       SEMj
       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. 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 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 backwarcI
       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.

[1]Set values, like those employed in HPSG, are not supported by ALE. In the categorial grammar in the appendix, they are represented by lists and treated by attached procedures for union and selection.

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 treat these two feature structures as being identical, as it does not conflate a cyclic structure with its infinite unfolding.

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:

```
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]

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

[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 $O(n^2)$ in the number of types.**

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.

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 / is defined in *F* then / 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

sign               sign
SUBJ agr           SUBJ CO] agr
     PERS first         PERS first
     NUM plu    <       NUM plu
OBJ agr            OBJ CO]
    PERS first
    NUM plu

false              false              [1] false
ARGl false      <  ARGl CO] false  <  ARGl [l]
     ARGl false         ARGl CO]
```

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.

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 CO] agr
     PERS 1st     OBJ [0]              PERS first
OBJ agr                               NUM plu
    NUM plu                        OBJ Co]

t           t           t
F [0] t  +  F t      -  F Cl] t
G CO]         F [1]       FCl]
              G Cl]       G Cl]

agr         +  agr          =  *failure*
PERS first     PERS second
```

[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 *O(n* • acifc""[1](n)), where *ack*~[1] is the inverse of Ackermanns 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.**

```
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.

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 a non-empty list can not 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.

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
```

[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, which are themselves left implicit. Furthermore, the `Tag` is used for forwarding and dereferencing during unification.

```
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:

```
functional sub [forward,backward]
           intro [argisynsem,
                  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

[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 can not be unique.

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 axe 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].
  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:xl.
    c sub []
```

```
        intro [fry,
               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 [argl:individual].
    unary.prop sub [].
    transitive.prop sub [binary.prop,ternary.prop]
                    intro [arg2:individual].
      binary.prop sub [].
      ternary.prop sub []
                   intro [axg3:individual].
```

In this case, unaxy 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.

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 two ways. First, we replace the notion of path sharing with the more compact and expressive notion of variable due to Smolka (19SSa). Second, we extend the language to types, following Pollard (in press).

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

```
<desc>  ::= <type>
          I <variable>
          I (<feature>:<desc>)
```

```
I (<desc>,<desc>)
I (<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 axound 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. 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.

Standard assumptions about operator precedence and association are followed by ALE, allowing us to omit most of the parentheses in descriptions. In particular, feature selecting colon binds the most tightly, followed by conjunction and then by disjunction. Furthermore, conjunction and disjunction are left-associative, while te 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)
```

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 <*j*> *entails* a description *xj)* if every structure satisfying <*j*> also satisfies *rp.* 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 (axgl: a, binary.proposition) would be slower than (binary-proposition,argl:a), since binary.proposition entails the existence of the feature argl, but not conversely.

At run-time, ALE computes a representation of the most general feature structure which 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 neJ.ist 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 neJList 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 be how large the structures become.[6] 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.[7]

[6]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.

[7]It corresponds so closely with non-determinism that satisfiability of descriptions with

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,fern].
    masc sub [].
    fern 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;fern. 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 eaxly in the computation.

disjunctions is NP-complete. Furthermore, the algorithm employed by ALE may produce up to 2^n satisfiers for a description with n disjunctions.

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
   synm,
   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 Q 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:Rl,conj 2:R2),Ind3)
```

would be equivalent to the description:

```
synm,
sem: (body: (and,conj 1 :Rl,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,conjl:Rl,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:  0 np(Name) ,
   (8 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.

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

[HIT]               (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 Q. 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.

Chapter 4

Definite Constraints

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 constraints 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

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

with attribute-value logic descriptions. The language in which clauses axe expressed in ALE is given in BNF as:

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

<literal> ::=  <pred_sym>
             I  <pred_sym>(<desc_seq>)

<desc_seq> ::= <desc>
             I <desc>,<desc_seq>

<goal> ::= true
         | <literal>
         | (<goal>,<goal»
         | (<goal>;<goal»
         | !
         | (\+ <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 axound the if operator, but none is required axound the conjunction comma, the disjunction semicolon, the cut symbol !, 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, literals can't 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.
```

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

```
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 axe 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.

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

```
member(X,[X|J)  if
  true,
member(X,LIXs]) 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].
      1 sub [].
      r sub [].
    glide sub [y,w].
      y sub [].
      w sub [].
```

```
vowel sub [a,e,i]
  a sub [].
  e sub [].
  i sub [].

less_sonorousjbasic(nasal,liquid) if true,
less.sonorous_basic(liquid,glide) if true.
less_sonorous_basic(glide,vowel) if true.

less_sonorous(Ll,L2) if
  less_sonorous_basic(Ll,L2).
less_sonorous(Ll,L2) if
  less_sonorous_basic(Ll,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 steins 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, negation 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.

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

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

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 negations 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.

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 uses a bottom-up active chart parser rather than encoding grammars direclty 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 one for describing grammar rules. We consider these components in turn, after a discussion of the parsing algorithm.

Parsing

It is not necessary to fully understand the parsing algorithm employed by ALE to exploit its power for developing grammars. But for those users concerned with efficiency and writing grammars with procedural attachments, it is crucial information.

The ALE system employs a bottom-up active chart parser which has been tailored to the implementation attribute-value grammars in Prolog. The single most important fact to keep in mind is that rules are evaluated from left to right. Most of the implementational considerations follow from this rule evaluation principle and its specific implementation in Prolog.

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 inactive edges are recorded. Inactive edges are purely dynamic structures, existing only locally to exploit Prolog's copying and backtracking schemes. The benefit of paxsing from right to left is that when an active edge is proposed by the bottom-up rule, every inactive edge it might need to be completed has already been found. 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, the bottom-up rule, and the 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.

Rules can incorporate definite clause goals before, between or after category specifications. These goals are evaluated when they are found. For instance, if a goad 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. As things stand, ALE does not keep track of the parse tree. 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.

The current version of ALE has no mechanism for detecting duplicate edges. Thus there is no mechanism 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. Furthermore, it might be the case that there is redundancy, in the sense that there are two derivations of the same category. ALE is also unable to detect this situation. This strategy was followed rather than the standard one which checks for subsumption when an edge is added, because it was felt that most grammars do not have any spurious ambiguity. Most unification-based grammars incorporate some notion of thematic or functional structure representing the meaning of a sentence. In these cases, most structural ambiguities result in semantic ambiguities. Thus it would actually slow the algorithm down to constantly check for a condition that never occurs. Future versions of ALE should allow the user to set a flag which determines whether spurious ambiguity and redundancy is captured during parsing.

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

```
John——>
   1D pn(j).

pn(Name) macro
   synsem: Q np(Name),
   Q quantifier.free.

np(Ind) macro
   syn:np,
   semilnd.

quantifier_free macro
   qstore: [].
```

Read declaratively, this rule says that the word j ohn has as its lexical category the most general satisfier of the description <3 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 axe useful as a method of organizing lexical information to keep it consistent across lexical entries. The lexical entry for the word runs is:

```
  runs——> <B iv((run,runner:Ind),Ind).

  iv(Sem,Arg) macro
    synsem:(backward,
            arg: Q np(Arg),
            res:(syn:s,
                 sem:Sem)),
    <0 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. Thus the first three lexical entries, taken together, are identical to the fourth:

```
  bank——>
    syn:noun,
    sem:river_bank.
  bank——>
    syn:noun,
```

```
    sem:money.bank.
bank___>
    synrverb,
    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 which 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 lexicons, as it relies on Prolog's hashing mechanism to actually look up a lexical entry for a word during bottom-up parsing.

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(Quant) macro
    synsem:(forward,
            arg:(syn:(n,
                      num:plu),
                 sem:(body:Restr,
                      ind:Ind)),
            res:(syn:(np,
                      num:plu),
                 semrlnd),
      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 is used in ALE. The reason for this is that these categories must be inserted 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 will produce an inactive edge at every node in the chaxt, then match the forward application rule scheme and search every edge to its right looking for a nominal complement. Because 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* and *C2* as empty categories, and have a rule that allows a *C* to be constructed from a *C* and a C2, then *C* will act as an empty category, as well. These combinations of empty categories are computed at run-time,

and may be a processing burden if they apply 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.

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 line initially somewhere in 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 are more than one such specification in an input file, the last one will be the one that is used.

The format for lexical rules is as follows:

```
<lex_rule> : := <lex_rule_name> lex.rule <lex_rewrite>
                                morphs <raorphs>.

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

<morphs> : := <morph>
            I <morph>, <morphs>
```

```
<morph> ::= (<string_pattern>) becomes (<string_pattern>)
          I (<string_pattern>) becomes (<string_pattern>)
                               when <prolog_goal>

<string_pattern> ::= <atomic_string_pa.ttern>
                   I <atomic_string_pattern>, <string_pattern>

<atomic_string_pattern> ::= <atom>
                          I <var>
                          I <list(<var_char>)>
<var_char> ::= <char>
             I <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,
   numrsing)
   **> (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 mutliple 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:

[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.

```
[g,o,o,s,e] becomes [g,e,e,s,e],
[k,e,y] becomes [k,e,y,s],
(X,[n,a,n]) becomes (X,[m,e,n]),
(X,F) becomes (X,F,[e,sj) 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:

```
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:[Gils])
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 defaulty 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.

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> ::= <mle_name> rule <desc> ==> <nile_body>.

<nile_body> ::= <mle_clause>
              I <mle_clause>, <mle_body>

<rale_clause> ::= cat> <desc>
                I goal> <goal>
```

The <rale_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 an ALE goal. 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,
   semrNPSem),
cat>
  (syn:vp,
   agr:Agr,
   semrVPSem).
```

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.

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

Even though the parsing proceeds from right to left, rules are evaluated from left to right, so that the descriptions of daughter categories are evaluated in the order in which they are specified. This is significant when considering goals that might be interleaved with searches in the chart for consistent daughter categories.

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 axe 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. 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 shaxed 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.

Finally, it should be kept in mind that the mother category description is evaluated for most general satisfiers only after the categories and goals in the body of the rule have been solved.

A more complicated rule, drawn from the categorial grammar in the appendix, and involving a non-trivial goal, is as follows:

```
backvard.application rule
(synsem:Z,
 qstore:Qs)

cat>
  (synsem:Y,
   qstore:Qsl),
cat>
  (synsem:(backward,
           arg:Y,
           res:Z),
   qstore:Qs2),
goal>
  append(Qsl,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 sequenced 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 Qsl and Qs2 are instantiated. The append(Qsl,Qs2,Qs) goal is then evaluated by ALE's definite clause resolution mechanism. 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 ALE definite clause goals, and thus may include disjunctions, conjunctions, negations, cut, etc., all of which will be evaluated normally in the context of a phrase structure rule.

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.

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 or Quintus Prolog. As the system was developed with SICStus, which is meant to be compatible with Quintus, ALE should work with either of these Prolog compilers. It would be futile to run ALE with only a Prolog interpreter, as it would be slowed by at least two orders of magnitude. The local systems administrator should be able to provide help in 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.

File Management

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

I ?- 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(Vusers/carp/Prolog/ALE/ale.piO ., or a local abbreviation for it like compile(ale) . if the system is in a file named ale.pl in the local directory (SICStus, at least, can fill in the ".pi" suffix). 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 have 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.[1] For instance, the sub/intro

[1] **Unless the appropriate multifile declarations are made.**

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.

Compiling Programs

ALE can compile a program incrementally to some extent. In particular, the compiler is broken down into four primary components for compiling the type hierarchy, the attribute-value logic, the definite clauses and the grammar. Compiling the type hierarchy consists of compiling type subsumption, type unification and appropriateness specifications. 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. Compiling the grammar consists of compiling the lexicon, empty categories, rules and lexical rules. Macros are not compiled, but are rather interpreted during compilation.

There is one predicate `compile_gram/1` which 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.

The following predicates are avaiable 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	Clause
compile_grammar	nothing	*	
compile_sig	nothing	*	
compile_sub_type		*	sub
compile_unify_type	compile_sub_type		
compile_approp	compile_unify_type	*	intro
compile_logic			
compile_add_to_type	compile_sig		
compile_featval	compile_add_to_type		
compile_u	compile_sig		
compile_dcs	compile_logic	*	if
compile_grammar	compile_logic	*	

compile_lex	compile_logic	*————>
compile.empty	compile.logic	* empty
compile.rules	compile.logic	* rule

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_add_to_type and compiledeatval). 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 paxt of a program where the relevant clauses have not been asserted, an error will result.

Each of the lowest level commands generates a file in the directory from which Prolog was called, (add.to.type and featval actually generate two files each). These files contain Prolog source code that is then compiled to generate the run-time environment for ALE. Thus it is important to have write permission in the directory from which ALE is being called. While these files are in an ASCII format, they are not intended to be read by ALE users.

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.

Unfortunately, the ALE compiler is itself not very efficient, though it produces rather efficient code. Thus it is always a good idea to recompile as little as possible. The savings in time can be significant.

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 can not 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 standaxd 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: JC, bar:JC) 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.

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.

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:

```
I ?- type(Type).

Type = bot ?;

Type = cat ?;

Type = synsem ?

yes
```

Note that the prompt I ?- 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:

```
I ?- type(bot).

yes

I ?- 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:

```
I ?- 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:

```
I ?- unify_type(Tl,T2,T).
```

The interpretation here is that Tl 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:

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

Feature introduction can be tested by:

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

which holds if feature F is introduced at type T.

Finally, the inherited appropriateness function can be tested by:

```
I ?- 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 has been performed). An example of the show_type/l query is as follows:

```
I ?- 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.

Evaluating Descriptions

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

```
I ?- 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:eJList 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:

```
I ?- 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.

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:*

```
I ?- no_write-type(T) .
```

After no-vrite-typeCD is called, the type *T* will no longer be displayed during printing. To restore the type *T* to printed status, use:

```
I ?- 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:

```
I ?- 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:

```
I ?- no-writeJeat(F) .
```

To restore printing of feature F, use:

```
I ?- 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.

```
I ?- write_feats.
```

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:

```
I ?- 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:

```
I ?- 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:

```
I ?- 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.

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:

```
I ?- 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.

Empty lexical entries can be displayed using:

```
I ?- empty.

EMPTY CATEGORY:
    cat
    QSTORE ne_list_quant
           HD some
              RESTR [0] proposition
              SCOPE proposition
              VAR [1] individual
           TL eJList
    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:

```
I ?- rule forward.application.

RULE: forward.application

MOTHER:
```

```
  cat
  QSTQRE [4] list.quant
  SYNSEM [0] synsem

DAUGHTERS/GOALS:

CAT cat
     QSTORE [2] list.quant
     SYNSEM forward
            ARG [1] synsem
            RES [0]

CAT  cat
     QSTQRE [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:

```
I ?- 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.

Executing Grammars

In this section, we consider the execution of ALE phrase structure grammars. The primaxy predicate for parsing is illustrated as follows:

```
I ?- rec [John,hits,every,toy] .

STRING:
0 John 1 hits 2 every 3 toy 4

CATEGORY:
cat
QSTDRE 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 which indicate positions in the string for later use in inspecting the chart directly. 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:

```
I ?- 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 which it can find for the input string. This allows sentence fragments and phrases to be analyzed, as in:

```
I ?- 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.
```

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:

```
I ?- 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

ANOTHER? n.
```

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 which are not available after they have been evaluated.

Using edge/2 it is possible to debug grammars by seeing how fax analyses got and inspecting analyses of substrings. In the current version of ALE, this is all that is provided in the way of debugging. Future releases should be greatly improved along this dimension.

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).

AiVKaci, 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., and Nasr, R. (1986b). LOGIN: A logical programming language with built-in inheritance. *Journal of Logic Programming,* 3:1S7—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.

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.

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(l):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).

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 prerequisite.

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. (in press). *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.

Smolka, G. (1988a). A feature logic with subsorts. LILOG-REPORT 33, IBM - Deutschland GmbH, Stuttgaxt.

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. (19S6). *The AH of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Massachusetts.

Best general introduction to logic programming in Prolog.

Appendix A

Sample Grammars

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] .
        1 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 [hdrbot,
                   tlilist].
    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].
```

7. Rules

7. =====

```
word.schema.rec rule
(word,
 syllables: [SyllableI Syllables] ,
 first:Firstl,
 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:[VowellSegsl],
 first:Vowel,
 last:Last)

cat> (vowel,Vowel),
cat> (coda,
     segments:Segsl,
     last:Last).

cv.syllable rule
(syllable,
 syllable:Segs,
```

```
 first:First,
 last:Vowel)

cat> (onset,
     segments:Segsl,
     first-.First),
cat> (vowel,Vowel),
goal> append(Segsl,[Vowel],Segs) .

cvc.syllable rule
(syllable,
 syllable:Segs,
 first:First,
 last:Last)
===>
cat> (onset,
     segments:Segsl,
     first:First),
cat> (vowel,Vowel),
cat> (coda,
     segments:Segs2,
     last:Last),
goal> append(Segsl, [VowelISegs2],Segs) .

consonant_cluster_base rule
(consonant.cluster,
 segments:[Consonant],
 first:Consonant,
 last:Consonant)

cat> (consonant,Consonant).

onset rule
(onset,
 segments:[Consonant11Consonants],
 first:Consonant1,
 last:Consonant3)

cat> (consonant,Consonant1),
cat> (onset,
     segments:Consonants,
     first:Consonant2,
```

```
     last:Consonant3),
goal> less.sonorous(Consonant1,Consonant2) .

coda rule
(coda,
 segments:[Consonant11Consonants],
 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.
1___> 1.
r -> r.
y___> y.
w___> w.
a___> a.
e -> e.
i -> i.
o___> o.
u___> u.
```

7. 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(Ll,L2) if
  less_sonorous_basic(Ll,L2).
less_sonorous(Ll,L2) if
  less_sonorous_basic(Ll,L3),
  less.sonorous(L3,L2).

append([],Xs,Xs) if true.
append([X|Xs],Ys,[X|Zs]) if
  append(Xs,Ys,Zs).
```

Categorial Grammar with Cooper Storage

```
7. 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,
                    bodyrproposition].
    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,
                   scoperproposition].
        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____>
  0 tv(hit).


%. Grammar
%. =======

forward.application rule
(synsem:Z,
 qstore:Qs)
===>
cat>  (synsem:(forward,
       arg:Y,
       res:Z),
       qstorerQsl),
cat>  (synsem:Y,
       qstore:Qs2),
goal> append(Qsl,Qs2,Qs).


backward.application rule
(synsem:Z,
 qstore:Qs)
===>
cat>  (synsem:Y,
       qstore:Qsl),
cat>  (synsem:(backward,
       arg:Y,
       res:Z),
       qstore:Qs2),
goal> append(Qsl,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,
                     argl:X),
               ind:X)),
  <8 quantifier.free.

gdet(Quant)  macro
  synsem:(forward,
          arg:  0 n(Restr,Ind) ,
          res:  0 np(Ind)),
  qstore:[Q quant(Quant,Ind,Restr)].

quant(Quant,Ind,Restr) macro
  (Quant,
   var:Ind,
   restr:Restr).

adj(Rel) macro
  synsem:(forward,
          arg:  Q n(Restr,Ind),
          res:  Q n((and,
                     conjl:Restr,
                     conj2:(Rel,
                            argl:Ind)),
                     Ind)),
  0  quantifier.free.

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

```
       ind:Ind).

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

pn(Name) macro
  synsem: Q np(Name),
  ® quantifier.free.

iv(Sem,Arg) macro
  synsem:(backward,
          arg: Q 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)))),
  Q 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([qi|Qsl],q,[qi|qs2])  if
  select(Qsl,Q,Qs2).
```

Appendix B

Error and Warning Messages

Error Messages

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.

consistent *T* and *T-i* have multiple mgus *Ts*

Types Ti and *Ti* have the non singleton set *Ts* as their set of most general unifiers.

feature *F* multiply introduced at *Ts*

The feature *F* is introduced at the types in *Ts*, which are not comparable with one another.

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.

no lexical entry for *W*

Expression *W* is used, but has no lexical entry.

unsatisfiable lexical entry for *W*

Word *W* has a lexical entry which has no satisfying feature structure.

`invalid line <j> in rule`

A line of a grammar rule is neither a goal nor a category description.

description uses unintroduced feature *F*

A description uses the feature *F* which has not been defined as appropriate for any types.

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 7r

A path 7T of features uses undefined feature *F* in a description.

subtype 7\ used in *T<i* undeclared

Undefined type Z\ declared as subtype in definition of T_2.

***T*\ used in appropriateness definition of *T2* undefined**

Undefined type I\ used as value restriction in definition of *T2*.

***T* multiply defined**

There is more than one definition of type *T*.

multiple specification of *F* in definition of *T*

More than one restriction on the value of feature *F* is given in the definition of type *T*.

appropriateness cycle following path -*K* from *T*

There is a sequence of features TT which must be defined for objects of type *T* where the value must be of type *T*.

rule *R* has no 'cat>' specification

The grammar rule named *R* is empty in that it does not have any daughter specification.

Warning Messages

`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.

`no features introduced`

There are no appropriate features for any types.

`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.

`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 phrase structure rules found`

There were no phrase structure rules specified in the program.

`no definite clauses found`

There were no definite clause rules specified in the program.

Appendix C

BNF for ALE Programs

The following is a complete BNF grammar for ALE programs.

```
<desc>  ::= <type>
          | <variable>
          | (<feature>:<desc>)
          | (<desc>,<desc>)
          | (<desc>;<desc>)
          | <0 <macro_spec>

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

<macro_head>  ::= <macro_name>
                I <macro_name>(<var_seq>)

<macro_spec>  ::= <macro_name>

                I <macro_name>(<desc.seq>)

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

<literal>  ::=  <pred_sym>
             I  <pred.sym>(<seq(<desc>)>)
<goal>  ::= true
          | <literal>
          | (<goal>,<goal>)
          |
          | (<goal>;<goal>)
          |
            (\+ <goal>)

<lex.entry>  ::=  <word>——> <desc>.
```

```
<mle> ::=  <rale_name> rule <desc> ===> <rule_body>.

<rule_body> ::=  <rule_clause>
              I  <rule_clause>, <mle_body>

<rule_clause> ::=  cat> <desc>
                I  goal> <goal>

<lex_rule> ::= <lex.rule.name> lex.rule <lex_revrite>
                               morphs <morphs>.

<lex.rewrite> ::= <desc> **> <desc>
                I  <desc> •*> <desc> if <goEL>

<morphs> ::= <morph>
           I <morph>, <morphs>

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

<string_pattern> ::= <atomic.string.pattern>
                   I <atomic.string.pattern>, <string.pattern>

<atomic.string.pattern> ::=  <atom>
                           I  <var>
                           I  <list(<var_char>)>
<vax_char> ::= <char>
             I <var>

<seq(X)> ::= X
           I X, <seq(X)>

<empty.prod> ::= empty <desc>.

<type_spec> ::= <type> sub <list(<type>)>
              I <type> sub <list(<type>)>
                       intro <list(<feat>:<type>)>

<prog> ::= <prog.line>
         I <prog_line> <prog>
```

```
<prog_line>  ::= <type_spec>
               | <macro_def>
               | <empty.prod>
               | <clause>
               | <rule>
               | <lex_entry>
               | <lex.mle>
```