GENERALIZED LL(k) GRAMMARS FOR CONCURRENT LOGIC PROGRAMMING LANGUAGES

by

Harvey Abramson

Technical Report 87-32

October 1987

Generalized LL(k) Grammars

for

Concurrent Logic Programming Languages

Harvey Abramson
Department of Computer Science
University of British Columbia
Vancouver, B.C.
Canada

ABSTRACT

We examine the compilation of the LL(k) deterministic context free grammars to Horn clause logic programs and sequential and concurrent execution of these programs. In the sequential case, one is able to take advantage of the determinism to eliminate the generation of unnecessary backtracking information during execution of the compiled logic program. In the concurrent case, grammar rules are simply and directly translated to clauses of Concurrent Prolog, Parlog, or Guarded Horn Clause programs, allowing grammatical processing directly in the setting of committed or "don't care" nondeterminism. LL(k) grammar rules are generalized so that grammatical processing of streams involving derivations of infinite length is possible. A top-down analogue of Marcus's deterministic parser is a possible application of these generalized LL(k) grammars.

October 19, 1987

Generalized LL(k) Grammars

for

Concurrent Logic Programming Languages

Harvey Abramson
Department of Computer Science
University of British Columbia
Vancouver, B.C.
Canada

1. Introduction.

A grammar is a finite way of specifying a possibly infinite set of sentences of a language. A logic grammar is a grammar whose rules can be represented by or compiled to Horn clauses. A logic grammar thus has not only a declarative reading, specifying the sentences of a language, but also a procedural reading, permitting sentences of the language to be analysed or synthesized. Since the introduction of Metamorphosis Grammars 5) various kinds of logic grammars have been devised (see 1), 2), 7), 8), 9), 12) and 13)). Although the declarative reading of the Horn clauses corresponding to grammar rules in these various formalisms is quite general, the procedural reading has depended on the top down, left to right sequential execution strategy of Prolog, in which backtracking is used to simulate nondeterministic search.

Recently logic programming languages have been defined which directly exploit the possibilities of concurrency inherent in logic programming. The most successful such languages, Parlog 4), Concurrent Prolog 14), and Guarded Horn Clauses 16), however, make use of committed nondeterminism as a control strategy rather than full and/or parallelism. (See 6) for discussion of a common execution model for these languages.) The form of a clause in these languages, ignoring differences in notation, is roughly :

r(t1,...,tk) :- <guards> : <body>.

Both the guards and the body are a conjunction of goals. In attempting to evaluate a goal r(p1,...,pk), all clauses for the relation r are attempted in parallel. Here, attempted means matching the head of the clause and successful evaluation of the guards. From those clauses which are successfully attempted, one is selected, and the others are discarded. This is "don't care" or "committed"

nondeterminism: the discarded calls are not cared about, or, one is committed to a particular choice once made. In practice, the first attempt to succeed is chosen. In the languages mentioned there are also synchronization mechanisms for delaying calls to make sure that certain arguments are instantiated. In Concurrent Prolog this is done by annotating arguments as read-only; in Parlog, mode declarations specify which arguments are input or output arguments; in Guarded Horn Clauses, neither annotations nor mode declarations are used, but if during unification of the head or execution of the guards an attempt is made to bind a non-local variable, then execution suspends. During the attempt to evaluate a goal, any arguments, say in a guard, which are not instantiated when they should be, result in a suspension until the variable in question, shared by some other process, is more fully instantiated.

There is an obvious problem in parsing in a committed nondeterministic setting. From the productions which may be successfully attempted, the processor will select one, commit to it, and ignore all the others. This will obviously allow the derivation to continue one more step, but it may not allow the derivation to continue to a successful conclusion. For example, suppose we had the following productions for a nonterminal "x" and we were parsing in a setting of committed nondeterminism:

x ::= [].
x ::= a, b, c.

Suppose at some point in the parse, both productions were successfully attempted (assume empty guards) but that the processor had chosen the empty production to commit to. Even though the input may be parsed as an "a" followed by a "b" and a "c", the wrong production (always applicable because of the empty right hand side) will have been chosen and a parse will not be found.

Clearly, if there is to be any class of logic grammars for which there is a simple direct translation of grammar rules to concurrent logic program clauses in a setting of committed nondeterminism, then that production (clause) must always be selected which will allow a derivation to continue to a successful conclusion if one exists This will happen if at any time at most one production can be used to continue a derivation. Fortunately, there is a subclass of context free grammars, the LL(k) grammars, which provides a model for such a class of logic grammars. The class of LL(k) grammars consists of those unambigous context free grammars in which input is parsed top down from left to right with k-symbol lookahead. The lookahead enables one to uniquely determine which production is to be used in continuing a parse. If no production is applicable, then the input string is not in the language generated by the grammar. This class is deterministic in

the sense that it can be accepted by a deterministic pushdown automaton.

We shall use the following LL(1) grammar, taken from 3), first to show how deterministic grammars may be compiled as sequential logic programs and then, generalizing, compiled to concurrent logic programs with "don't care" nondeterminism (note the paradox: deterministic grammars compile easily into don't care nondeterministic logic programs!).

1.1. Sample grammar.

```
e ::= t,e_prime.

e_prime ::= "+",t,e_prime.
e_prime ::= [].

t ::= f,t_prime.

t_prime ::= "*",f,t_prime.
t_prime ::= [].

f ::= "a".
f ::= "(",e,")".
```

See the last section of this paper for comments on nondeterministic parsing.

1.2. The one character lookahead relation.

The following unit clauses define the one character lookahead relation for the sample grammar. The first argument to the predicate "lookahead" is a production, and the second is a list of characters which permit use of that production in a derivation. For example, the production "e::=t,e_prime" may be used if searching for an "e" and if the first unused character in the input string, the lookahead character, is either an "a" or a "(".

```
lookahead((e::=t,e_prime),"a(").
lookahead((e_prime::="+",t,e_prime),"+").
lookahead((e_prime::=[]),")?").
lookahead((t::=f,t_prime),"a(").
lookahead((t_prime::="*",f,t_prime),"*").
lookahead((t_prime::=[]),"+)?").
lookahead((f::="a"),"a").
lookahead((f::="(",e,")"),"(").
```

The "?" is used to mark the end of the input string. The "lookahead" predicate may be calculated following an algorithm given in 3)

and is easily specified in Prolog (although some version of "setof" must be used). The hand-coding of LL(k) grammars into Prolog clauses was very briefly mentioned in 15).

2. Compilation to sequential logic programs.

Although our principal motivation is to find a class of concurrent logic grammars, we begin with the compilation of deterministic grammars to sequential logic program clauses: the sequential case is itself interesting and gives the foundation of the method to be developed for the concurrent case.

In compiling LL(k) grammars to sequential logic programs we would like to take advantage of the determinacy of production use in a derivation. We shall do so by treating the lookahead examination as a guard on use of a clause compiled from a production. The logic program clauses generated from the above grammar will each have as their first goal a call to a predicate called "ll_guard". This is a predicate of arity 4: the first argument is the original production used to index into the lookahead predicate; the second argument is treated as a node in the tree representation of the derivation and is a function symbol of the form guard(X), recording the lookahead string X (see 1) for a description of Definite Clause Translation Grammars and the automatic formation of a derivation tree); the last two arguments represent the input string as a difference list. For the LL(1) case, the "ll_guard" predicate is specified as:

```
ll_guard(LookAhead,guard(X),[X|Xs]) :-
  lookahead(LookAhead,List),
  member(X,List),!.
```

This does not use up any characters in the input string but merely examines them. The cut, once the lookahead "List" has been accessed, and the first character of the input string has been shown to be a member of that list, is used to ensure that there will be no backup in trying to use any other productions. (A clever compiler might avoid generating choice points which would not be used.)

Here follows the set of clauses generated for the above sample grammar. The call to "ll_guard" is automatically inserted by the grammar compiler. The third argument to the function symbol "node" represents an empty set of semantic rules. See 1) regarding the semantic component of grammar rules.

```
e(node(e,[Guard,T,E_prime],[]),S1,S3) :-
  ll_guard((e::=t,e_prime),Guard,S1),
  t(T,S1,S2),
  e_prime(E_prime,S2,S3).
```

```
e_prime(node(e_prime,[Guard,[+],T,E_prime],[]),S1,S4) :-
  ll_guard((e_prime::=[+],t,e_prime),Guard,S1),
  c(S1,+,S2),
  t(T,S2,S3),
  e_prime(E_prime,S3,S4).

e_prime(node(e_prime,[Guard,[]],[]),S1,S2) :-
  ll_guard((e_prime::=[]),Guard,S1),
  S1=S2.

t(node(t,[Guard,F,T_prime],[]),S1,S3) :-
  ll_guard((t::=f,t_prime),Guard,S1),
  f(F,S1,S2),
  t_prime(T_prime,S2,S3).

t_prime(node(t_prime,[Guard,[*],F,T_prime],[]),S1,S4) :-
  ll_guard((t_prime::=[*],f,t_prime),Guard,S1),
  c(S1,*,S2),
  f(F,S2,S3),
  t_prime(T_prime,S3,S4).

t_prime(node(t_prime,[Guard,[]],[]),S1,S2) :-
  ll_guard((t_prime::=[]),Guard,S1),
  S1=S2.

f(node(f,[Guard,[a]],[]),S1,S2) :-
  ll_guard((f::=[a]),Guard,S1),
  c(S1,a,S2).

f(node(f,[Guard,['('],E,[')']],[]),S1,S4) :-
  ll_guard((f::=['('],e,[')']),Guard,S1),
  c(S1,'(',S2),
  e(E,S2,S3),
  c(S3,')',S4).
```

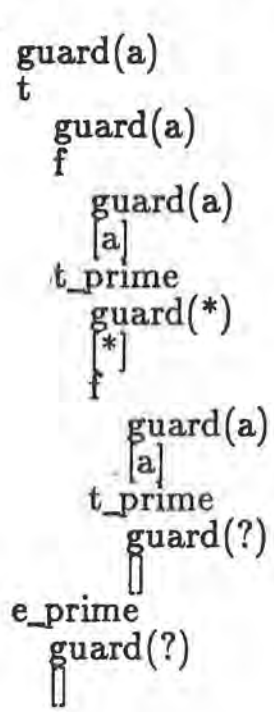
The predicate "c" is used to absorb a single terminal symbol:

```
c([X|Y],X,Y).
```

The controlling predicate "e" appends the endmarker, in this case, "?", calls the starting nonterminal of the grammar, and pretty prints the result:

```
e(Source) :-
  append(Source,[?],EndMarked),
  e(Guard,EndMarked,[?]),
  pretty(Guard).
```

For example, a call of "e("a*a")" yields:



3. Compilation to concurrent logic program clauses.

We shall illustrate the compilation of a deterministic grammar to a (don't care) concurrent logic program using the language Concurrent Prolog as a target; compilation to Parlog and GHC is similar, and we shall comment on this below. The basic idea is to turn the predicate "ll_guard" into a true guard on the generated clause and each nonterminal into a concurrent process. An attempt is made to reduce the generated clause only if the guard succeeds. The processes corresponding to nonterminals must be synchronized so that there is, in the LL(1) case, a character in the input string against which a guard may succeed or fail. The synchronization is accomplished by annotating the first of the two hidden arguments with a "?", the read-only annotation. If the input string is not yet sufficiently instantiated, the process delays until an input character has appeared. Here are the generated Concurrent Prolog clauses for our sample grammar. The commit operator is indicated by a ";".

```
e(node(e,[Guard,T,E_prime],[]),S1,S3) :-
  ll_Guard((e::=t,e_prime),Guard,S1);
  t(T,S1?,S2),
  e_prime(E_prime,S2?,S3).

e_prime(node(e_prime,[Guard,[]],[]),S1,S2) :-
  ll_Guard((e_prime::=[]),Guard,S1);
```

```
  S1? = S2.

e_prime(node(e_prime,[Guard,[+],T,E_prime],[]),S1,S4) :-
  ll_Guard((e_prime::=[+],t,e_prime),Guard,S1);
  c(S1?,+,S2),
  t(T,S2?,S3),
  e_prime(E_prime,S3?,S4).

t(node(t,[Guard,F,T_prime],[]),S1,S3) :-
  ll_Guard((t::=f,t_prime),Guard,S1);
  f(F,S1?,S2),
  t_prime(T_prime,S2?,S3).

t_prime(node(t_prime,[Guard,[]],[]),S1,S2) :-
  ll_Guard((t_prime::=[]),Guard,S1);
  S1? = S2.

t_prime(node(t_prime,[Guard,[*],F,T_prime],[]),S1,S4) :-
  ll_Guard((t_prime::=[*],f,t_prime),Guard,S1);
  c(S1?,*,S2),
  f(F,S2?,S3),
  t_prime(T_prime,S3?,S4).

f(node(f,[Guard,['('],E,[')']],[]),S1,S4) :-
  ll_Guard((f::=['('],e,[')']),Guard,S1);
  c(S1?,'(',S2),
  e(E,S2?,S3),
  c(S3?,')',S4).

f(node(f,[Guard,[a]],[]),S1,S2) :-
  ll_Guard((f::=[a]),Guard,S1);
  c(S1?,a,S2).
```

We use the following definition of "member" in the Concurrent Prolog setting:

```
member(X,[Y|_]) :-
  X = Y ; true.

member(X,[Y|Z]) :-
  dif(X , Y);
  member(X,Z).
```

The definition of the "ll_guard" predicate must be changed slightly since it examines the input string: it delays until there is a character in the input stream and the lookahead "List" has been supplied:

```
ll_guard(LookAhead,guard(X),[X|Xs]) :-
  lookahead(LookAhead,List),
  member(X?,List?).
```

The controlling predicate now calls on the Concurrent Prolog interpreter to solve the goal "e", with the endmarked input string, yielding if succesful, the derivation tree "T":

```
e(Source) :-
 append(Source,"?",EndMarked),
 solve(e(T,EndMarked,"?")),pretty(T).
```

In the case of Parlog, the compiler from grammars to Parlog clauses would have to annotate the processes corresponding to nonterminals with mode declarations which would insure that the last but one argument is an input variable. The predicate "ll_guard" would act as a guard to the generated Parlog clauses. This example has in fact been converted to Parlog by S. Gregory 10). Conversion of this technique to Guarded Horn Clauses should not be difficult.

4. Generalized deterministic grammars.

We have so far shown how LL(k) grammars could be compiled directly into either sequential or don't care nondeterministic logic programming languages. The class of LL(k) languages is in some respects a restrictive one: it does not include all context free grammars, for example. Thus, one could not take an arbitrary context free grammar and transform it into an LL(k) grammar and then generate an efficient logic program (efficient in the sense of not requiring backtracking). In practice, however, many languages (probably most programming languages) can be formulated using LL(k) grammars. It is fairly likely that any language (presumably, for convenience to the user, a fairly restricted subset of natural lanaguage) which might be used as a command language to a logic operating system could be specified by an LL(k) grammar for some small value of k. One would then be able to use the hardware of a concurrent logic machine to handle the necessary grammatical processing directly rather than relying on an attached sequential grammar processor for this task.

There are, however, some obvious generalizations of the techniques displayed above which get out of the restrictive LL(k) class.

Firstly, the guards may be generalized to do more than look at a certain number of characters of the input stream. Grammar productions could be written in the form:

nonterminal ::= <guards>: <right-hand side>

where the nonterminal "expands" to the right-hand side only if the

guards are succesfully evaluated. It would be up to the grammar writer to provide specifications of the guards so that the wrong production is not selected in the committed nondeterministic setting.

Secondly, the nonterminal symbols may be allowed, as in the case of Definite Clause Grammars or Definite Clause Translation Grammars, to have more arguments than just those automatically added by the compiler from grammar rules to logic programming clauses. This certainly takes the grammar rules out of the very restrictive LL(k) class, and even, as is well known from the DCG and DCTG experience, out of the class of context free grammars.

Thirdly, the right-hand side of extended grammar rules may also include communication with concurrent processes other than ones corresponding to nonterminal symbols. As in the case of DCGs and DCTGs, one might use the notation in the right-hand part of a grammar rule:

{ concurrent_process(A,...,Z) }

to specify that some concurrent process with shared variables "A" to "Z" must be successfully reduced for parsing to succeed.

One should also note that in the concurrent setting derivations may be of infinite length. The guards above are used to determine whether a derivation may be continued or not: they do not enforce any restrictions on the length of input. Thus, one may think of the generalized grammar rules as allowing one to do grammatical processing on streams rather than on finite strings. In this view, the grammar rules provide a notation for operating on what might be termed a "hidden stream": it is a mechanical task to generate the concurrent logic program clauses which make that stream explicit as above in the simple LL(1) case.

5. Related work and future investigations.

An analogy can be made between SLD-resolution over Horn clause programs and context-free derivations over context-free grammars. In place of grammar rules one has a program of Horn clauses; instead of replacing a nonterminal by the righthand side of a grammar rule whose lefthand side is that nonterminal, one seeks to unify a goal with the head of a clause, and if successful replace that goal either by the empty clause or by the body of the clause using the substitution derived from unification to instantiate variables. In the context-free grammar situation one tries to derive a sentential form without nonterminals; in SLD-resolution one tries to remove all subgoals, deriving the empty clause. Thus, SLD-resolution over Horn clause programs generalizes context-free derivations.

In this paper we have drawn an analogy between LL(k) grammars, a subclass of context-free grammars and commited choice nondeterministic concurrent logic programming. LL(k) grammars constitute a proper subclass of context-free grammars which can be parsed efficiently. The drawback to this class is that the grammars of the class are viewed as being less "natural" and less "expressive" than full context-free grammars. Given the analogy drawn between LL(k) grammars and committed choice concurrent programming languages, one hopes that the lack of "naturalness" and "expressiveness" characteristic of the grammars does not carry over to the programming languages. If it does, one might wish to investigate specifying problems in full and/or parallel logic and use some heuristic program transformation techniques to derive efficient, but possibly less "natural", committed choice concurrent programs. Note that committed choice concurrent programming languages, as well as generalized LL(k) grammars for such languages, have the full computing power of a Turing machine. The concepts "natural" and "expressive" hence are intuitive and must be placed within quotation marks.

Other approaches may be taken to parsing in languages such as Parlog, Concurrent Prolog and Guarded Horn Clauses. One could simply avoid the problem and drop into Prolog, making use of known classes of logic grammars for parsing; if all possible parses of a sentence were required, one could make use of various "all solutions" predicates for gathering the parses into a list. This method, although effective, is not very interesting as far as exploitation of concurrent logic programming languages is concerned.

The approach taken by Matsumoto in 11) with respect to parsing in a concurrent setting is an alternative to ours and is more general, but posssibly less efficient. Matsumoto's approach is to allow nondeterministic grammars and utilise a parsing method related to Chart Parsing and Earley Parsing. Potentially, all possible parses may be gathered and merged into a list of parses. This seems quite suited to non-deterministic natural language parsing but may be unnecessarily powerful when used with deterministic formal languages. Also, there may be problems when nonterminal symbols have arguments containing uninstantiated variables.

Having said that nondeterministic parsing may be more suitable for natural language parsing in general, we still think that deterministic concurrent parsing may be applied to natural language parsing in some cases. Marcus has reported considerable success with a deterministic bottom up parser which is essentially an LR(3) parser. It is tempting to speculate that a top down analogue of his parser can be as succesful.

On a less speculative level, one would like to have the grammatical processes in the concurrent setting as efficient and as inexpensive as possible. For much of the time the process corresponding to a

nonterminal may be inactive, coming alive only when some input had arrived on its input stream. These processes could presumably be efficiently implemented by having them do a busy wait or be blocked until activated.

Acknowledgment

This research was supported with the aid of an SUR grant from IBM Canada. An earlier version of this paper appears in Lecture Notes in Computer Science 225, Third International Conference on Logic Programming, London, England, July, 1986, edited by E. Shapiro, Springer.

References.

1) Abramson, H., Definite Clause Translation Grammars, Proceedings 1984 International Symposium on Logic Programming, Feb. 6-9, 1984, Atlantic City, New Jersey, pp. 233-241.

2) Abramson, H. Definite Clause Translation Grammars and the Logical Specification of Data Types as Unambiguous Context Free Grammars, Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Nov. 6-9, 1984.

3) Aho, A.V. & Ullman, S., Principles of Compiler Design, Prentice-Hall, 1977.

4) Clark, K. & Gregory, S., Parlog: Parallel Programming in Logic, Imperial College, Research Report DOC 94/4, London, 1984.

5) Colmerauer, A., Metamorphosis Grammars, in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

6) Crammond, J., An Execution Model for Committed-Choice Non-Deterministic Languages, Proceedings 1986 IEEE Symposium on Logic Programming, September 22-25, 1986, Salt Lake City, 148-158.

7) Dahl, V., More on Gapping Grammars, Proceedings of the International Fifth Generation Computer Systems Conference, Tokyo, 1984.

8) Dahl, V. & Abramson, H., Gapping Grammars, Proceedings - Second International Logic Programming Conference, Uppsala, Sweden, 1984.

9) Dahl, V. & McCord, M., Treating coordination in logic grammars, American Journal of Computational Linguistics, vol. 9, pp. 69-71,

1983.

10) Gregory, S. private communication.

11) Matsumoto, Y., A Parallel Parsing System for Natural Language Analysis, Lecture Notes in Computer Sc ience 225, Third International Conference on Logic Programming Proceedings, London, England, July, 1986, edited by E. Shapiro, Springer, pp. 396-409.

12) McCord, M., Modular Logic Grammars, Proceedings - Association for Computational Linguistics Conference, July, 1985.

13) Pereira, F.C.N. & Warren, D.H.D, Definite Clause Grammars for Language Analysis, Artificial Intelligence, vol. 13, pp. 231-278, 1980.

14) Shapiro, E.Y., A subset of Concurrent Prolog and its interpreter, Technical Report TR-003, ICOT, Tokyo, 1983.

15) Stabler, E., Deterministic and bottom-up parsing in Prolog, Proceedings, AAAI, pp. 383-386, 1983.

16) Ueda, K., Guarded Horn Clauses, ICOT Technical Report TR-103, Institute for New Generation Computer Technology, Tokyo, June, 1985.