Preference Logic Grammars[1]

Kannan Govindarajan
Bharat Jayaraman

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
E-Mail: {bharat,govin-k}@cs.buffalo.edu

Surya Mantha

System Sciences Laboratory
Building 128
Xerox, Webster Research Center
Webster, NY 14580
E-Mail: mantha@wrc.xerox.com

Abstract

Preference Logic Grammars (PLGs) are introduced in this paper as a declarative means of resolving ambiguity in Logic Grammars. An application of PLGs of special interest is optimal parsing, which is an extension of parsing wherein costs are associated with the different (ambiguous) parses of a string and the preferred parse is the one with least cost. Many problems can be viewed as optimal parsing problems, e.g., code generation, document layout, *etc.* We show that PLGs are an extension of Definite Clause Translation Grammars (DCTGs). Just as DCTGs can be directly translated into Constraint Logic Programs (CLPs), PLGs can be translated into Preference Logic Programs (PLPs). The computational model for preference logic programs keeps track of alternative proof paths (parses) and prunes paths that are not optimal. We also consider the problem of incremental optimal parsing, motivated by the document layout application: We describe how to augment the computational model to support restricted incremental querying, which captures changes to the input that an editor for documents might make. Our proposed incremental strategy reuses portions of the computation of the previous optimal parse to construct the new optimal parse.

[1]Technical Report TR 94-27, Department of Computer Science, SUNY-Buffalo, June 1994. This TR is a draft. Comments are welcome, and may be sent to any of the authors.

1 Introduction

Context-Free Grammars are a natural formalism for specifying the syntax of a language, while Logic is generally the formalism of choice for specifying semantics (meaning). Logic Grammars combine these two formalisms, and can be used for specifying syntactic and semantic constraints in a variety of parsing applications, from programming languages to natural languages [AD89]. Several forms of Logic Grammars have been researched over the last two decades. Basically, they extend Context-Free Grammars in that they generally allow:

- the use of arguments with nonterminals to represent semantics, where the arguments may be terms of some logic;
- the use of unification to propagate semantic information;
- the use of logical tests to control the application of production rules.

An important problem in these applications is that of resolving ambiguity—syntactic and semantic. An example of syntactic ambiguity is the "dangling else" problem in programming language syntax, while an example of semantic ambiguity is that of "quantifier scoping" in natural languages. Traditionally, ambiguity in these applications is resolved in some *ad hoc* manner; for example, ambiguity in the "dangling else" problem is resolved during semantic analysis phase of a compiler, by pairing up each "else" with the closest previous unpaired "else". In this paper, we provide a declarative means of specifying the criteria to be used in resolving syntactic and semantic ambiguity in a logic grammar.

Of special interest to us in this paper is the problem of *optimal parsing*. This problem is an extension of the standard parsing problem in that costs are associated with the different (ambiguous) parses of a string, and the preferred parse of the string is the one with least cost. Many applications such as optimal layout of documents [BMW92], code generation [AGT89], *etc.*, can be viewed as optimal parsing problems. We show in this paper how the criteria for laying out paragraphs—which a formatter such as TEXor LATEXwould use [KP81]—can be specified declaratively using a logic grammar.

We develop our approach starting from Definite Clause Translation Grammars (DCTGs) [Abr84]. DCTGs are a particular kind of Logic Grammar, and can be viewed as the logical counterpart of *attribute grammars* [Knu68], since they allow a clear separation between the syntactic and semantic components of the specification. We show in this paper that a simple extension of DCTGs provides a natural means of specifying ambiguity resolution and optimal parsing problems. We refer to the resulting class of grammars as *Preference Logic Grammars* (PLGs). Just as DCTGs can be translated into Constraint Logic Programs [JL87, vH89, JM], PLGs can be translated into Preference Logic Programs [GJM94]. The operational semantics of Preference Logic Programs constructs the optimal parse by keeping track of alternative parses and pruning parses that are suboptimal.

Motivated by the document layout application and the language-based editors of [Rep84], we address the problem of incremental optimal parsing, whose natural application is a WYSIWYG document editor: The updates that a document editor makes to a document is modelled in the PLP framework by a sequence of queries to the program that is the result of translating the PLG. Consecutive queries in this sequence have a specific relationship that captures the change made to the document by the editor. Our incremental algorithm reuses portions of the previous optimal parse to construct the new optimal parse. We also describe how under certain reasonable monotonicity conditions, we can make both the optimal parsing algorithm and the incremental version more efficient.

The rest of this paper is organized as follows: Section 2 discusses Definite Clause Translation Grammars [AD89] and their use in representing document structures; section 3 introduces Preference Logic Grammars, shows their use for ambiguity resolution and optimal parsing, and briefly discusses the translation of PLGs into Preference Logic Programs; section 4 discusses the operational semantics of Preference Logic Programs for computing the optimal parse; section 5 discusses how to modify the operational semantics to support incremental optimal parsing; and, finally, section 6 presents conclusions and the current status of the work.

2 Definite Clause Translation Grammars

DCTGs were introduced by Abramson [Abr84], and can be viewed as a logical counterpart of attribute grammars [Knu68]. In attribute grammars, syntax is specified by context-free rules, and semantics are specified by attributes attached to nonterminal nodes in the derivation trees and by function definitions that define the attributes. Attribute grammars and DCTGs can be readily translated into Constraint Logic Programs over some appropriate domain [JL87, vH89, JM]. It may be noted that we need Constraint Logic Programs rather than Definite Clause Programs because we are interested in interpreting the functions defining the attributes over an appropriate domain.

DCTG rules augment context-free rules in two ways: (1) they provides a mechanism for associating with nonterminals logical variables which represent subtrees rooted at the nonterminal; and (2) they provide a mechanism for specifying the computation of semantic values of properties of the trees. For example, we associate a logical variable `N` with a nonterminal `nt` by writing:

```
nt^^N
```

The logical variable `N` will be eventually instantiated to the subtree corresponding to the nonterminal `nt`. In order to specify the computation of some semantic value `X` of a property `prop` of the tree `N`, we write:

```
N^^prop(X)
```

The following is a simple DCTG from [AD89] that specifies the grammar and semantics of bitstrings.

```
bit ::= ''0''
<:>
bitval(0,_).

bit ::= ''1''
<:>
bitval(V,Scale) ::- V = 2^Scale.
```

```
bitstring ::= []
<:>
length(0),
value(0,_).

bitstring ::= bit^^B, bitstring^^B1
<:>
(length(Length) ::- B1^^length(Length1),
                    Length = Length1 +1),
(value(Value,ScaleB) ::- B^^bitval(VB,ScaleB),
                         S1 = ScaleB - 1,
                         B1^^value(V1,S1),
                         Value = VB + V1).

number ::= bitstring^^B
<:>
value(V) ::- B^^length(Length),
             S = Length - 1,
             B^^value(V,S).
```

The DCTG formalism differs from attribute grammars in the following ways:

1. The semantics is captured by associating a set of definite clauses to nonterminal nodes of a derivation tree.
2. No distinction is made between inherited and synthesized attributes in the notation. The use of logical variables that unify with parse trees eliminates the need for this distinction.

2.1 From DCTGs to Logic Programs

DCTGs can be directly translated into Constraint Logic Programs [Mal]: Each clause in the CLP program is derived from a grammar rule and the predicate symbols in the program are derived from nonterminal symbols in the grammar. We associate three extra arguments with each nonterminal symbol to construct parses automatically and to represent the difference list of characters being analyzed or synthesized. Each node in the parse tree constructed by the program is represented by a term of the following kind:

node(Nonterminal, List_of_Subtrees, Semantics)

where Nonterminal is the name of the nonterminal labeling the node, List_of_Subtrees is the list of subtrees immediately dependent on the node and Semantics is a representation of the properties of the node.

Translating the simple DCTG example above gives us the following CLP program:

```
bit(node(bit,[[0]],[Bitval]),Start,End,Scale) :-
        consume(Start,'0',End),
        bit_bitval([[0]],Bitval,Scale).
bit(node(bit,[[1]],[Bitval]),Start,End,Scale) :-
        consume(Start,'1',End),
        bit_bitval([[1]],Bitval,Scale).


bitstring(node(bitstring,[],[Value,Length]),Start,Start,Scale) :-
        bitstring_length([],Length),
        bitstring_value([],Value,Scale).
bitstring(node(bitstring,[BitTr,BitstrTr],[Value,Length]),Start,End,Scale) :-
bit(BitTr,Start,X,Scale),
bitstring(BitstrTr,X,End,_),
        bitstring_length([BitTr,BitstrTr],Length),
        bitstring_value([BitTr,BitstrTr],Value,Scale).

number(node(number,[BitstrTr],[Value]),Start,End) :-
        bitstring(BitstrTr,Start,End,_),
        number_value(BitstrTr,Value).
```

The definitions of predicates such as `number_value` are the same as in the DCTG. The translation from a DCTG to a CLP outlined here is representative of the general translation scheme. The reason for insisting on translating to CLP programs rather than definite clauses is that we want to interpret function symbols such as + and predicate symbols such as = over appropriate domains.

2.2 Specifying Document Structures using DCTGs

We now demonstrate the usefulness of DCTGs for specifying document structures starting from the attribute grammars specifications of [BMW92]. Logically, a paragraph is a sequence of lines where each line is a sequence of words. This view of a paragraph can be captured by the following context free grammar with regular right hand sides as in Backus-Naur Form grammars:

<para> ::= <line> +

<line> ::= [unit]+

Knuth and Plass [KP81] describe how to lay out a sequence of words forming a paragraph by computing the badness of the paragraph which depends on the badness of the lines that make up the paragraph. The badness of a line is determined by the properties of the line such as the total width of the characters that make up the line, the number of white spaces in the line, the stretchability and shrinkability of the white spaces, the desired length of the line, *etc.* [KP81] insists that each line in the paragraph be such that the ratio of the difference between actual length and the desired length and the stretchability or shrinkability (the adjustment ratio) be bounded. These can be captured by the attribute grammar below, which associates attributes with the nonterminals <para> and <line> and provides equations that define the attributes of each nonterminal.

<para> ::= <line> +

<line>_NUM(<line>).paralast := true

<para>.badness = SUM_i <line>_i.badness

<line> ::= [unit]+

<line>.naturallength = SUM_i [unit]_i.width + (*interwordnaturalwidth* X NUM([unit]))

<line>.difference = <line>.desiredlength - <line>.naturallength

<line>.adjustment = if <line>.difference > 0 then <line>.difference / <line>.stretchability
else <line>.difference / <line>.shrinkability

<line>.badness = | <line>.adjustment|*3

<line>.GUARD = *lineshrinkbound* < <line>.adjustment < *linestretchbound*

Other attributes such as stretchability and shrinkability can be defined similarly. Identifiers that are enclosed in *, *e.g.*, *lineshrinkbound*, are constants. Functions SUM_i and NUM compute the sum of and number of relevant objects in the parse tree rooted at the appropriate nonterminal. Clearly, the attribute grammar captures the essence of the attributes necessary to layout a paragraph as described in [KP81].

The document structure described above can be written in the DCTG formalism in a very natural way. The underlying grammar in the example above can be equivalently written as follows:

<para> ::= <line>
<para> ::= <line> <para>
<line> ::= [unit]
<line> ::= [unit] <line>

Augmenting this grammar with attribute definitions gives us the following DCTG that is equivalent to the attribute grammar above:

```
para ::= line^^Line
<:>
badness(Badness) ::- Line^^paralast(true),
                     Line^^badness(Badness).
para ::= line^^Line, para^^Para
<:>
badness(Badness) ::- Line^^paralast(false),
                     Line^^badness(Linebadness),
                     Para^^badness(Parabadness),
                     Badness = Linebadness + Parabadness.

line ::= ‘‘unit’’
<:>
(naturallength(Naturallength) ::- width(unit,Width),
                                  Naturallength = Width + interwordshrinkability),
(difference(Difference) ::- desiredlength(Desiredlength),
```

```
                            naturallength(Naturallength),
                            Difference = Desiredlength - Naturallength),
(adjustment(Adjustment) ::- difference(Difference),
                            Difference > 0,
                            stretchability(Stretchability),
                            Adjustment = Difference/Stretchability),
(adjustment(Adjustment) ::- difference(Difference),
                            Difference =< 0,
                            shrinkability(Shrinkability),
                            Adjustment = Difference/Shrinkability),
(badness(Badness) ::- adjustment(Adjustment),
                      absolute(Adjustment AbsAdjust),
                      Badness = AbsAdjust*3),
(guard(Guard) ::- adjustment(Adjustment),
                  Adjustment < lineshrinkbound,
                  Adjustment > linestretchbound).

line ::= ``unit'', line^^Line
<:>
(naturallength(Naturallength) ::- width(unit,Width),
                                  Line^^naturallength(Length1),
                                  Length2 = Length1 + Width,
                                  Naturallength = Length2 + interwordshrinkability),
(difference(Difference) ::- desiredlength(Desiredlength),
                            naturallength(Naturallength),
                            Difference = Desiredlength - Naturallength),
(adjustment(Adjustment) ::- difference(Difference),
                            Difference > 0,
                            stretchability(Stretchability),
                            Adjustment = Difference/Stretchability),
(adjustment(Adjustment) ::- difference(Difference),
                            Difference =< 0,
                            shrinkability(Shrinkability),
                            Adjustment = Difference/Shrinkability),
(badness(Badness) ::- adjustment(Adjustment),
                      absolute(Adjustment, AbsAdjust),
                      Badness = AbsAdjust*3),
(guard(Guard) ::- adjustment(Adjustment),
                  Adjustment < lineshrinkbound,
                  Adjustment > linestretchbound).
```

3 Preference Logic Grammars and Preference Logic Programs

We now introduce a simple extension of DCTGs called Preference Logic Grammars (PLGs), and show how they can be translated into Preference Logic Programs [GJM94].

3.1 Preference Logic Grammars

The grammars specifying document structures, such as the one in the previous section, are extremely ambiguous. Given a description of a document and a sequence of words representing its content, we are interested in a parse that has a particular property. For instance, we may be interested in the parse of a sequence of words from <para> that has the least badness. [BMW92] augmented attribute grammars with minimization directives that specified which attributes had to be minimized in the preferred parse. Similarly, we extend DCTGs with statements that specify the preferred parse. For instance, in the line-breaking example above, the preferred parse is specified by augmenting the definition of para as follows:

```
para ::= line^^Line
<:>
badness(Badness) ::- Line^^paralast(true),
                     Line^^badness(Badness).
para ::= line^^Line, para^^Para
<:>
badness(Badness) ::- Line^^paralast(false),
                     Line^^badness(Linebadness),
                     Para^^badness(Parabadness),
                     Badness = Linebadness + Parabadness.

para^^Para^^badness(B) --> Pf(para^^Para1^^badness(B1), B1 < B).
```

The symbol Pf is the monadic modal operator of preference introduced in [Man91]. Such rules are called preference rules and specify that the preferred parse from the nonterminal para is the one whose badness property is the least in the ordering <.

We now turn to an example illustrating how to specify the criteria for resolving ambiguity in context free grammars. Consider the following grammar which exhibits the "dangling else" ambiguity:

<stmtseq> ::= <stmt> | <stmt> ; <stmtseq>
<stmt> ::= <assign> | <ifstmt>
<assign> ::= <var> := <expr>
<ifstmt> ::= if <cond> then <stmtseq> | if <cond> then <stmtseq> else <stmtseq>

Given a string of the form:

if cond1 then if cond2 then assign1 else assign2

there are two possible parses (parentheses are used below to indicate grouping):

if cond1 then (if cond2 then assign1 else assign2)
if cond1 then (if cond2 then assign1) else assign2

A natural way to resolve the ambiguity is to express our preference for one parse over the other; the resulting solution is far more succinct than rewriting the grammar to avoid ambiguity. For the purpose of this example, it suffices to encode the above grammar using a Definite Clause Grammar (DCG) rather than as a DCTG. Assuming definitions for the nonterminals `expr`, `cond`, and `var`, the ambiguous DCG is as follows:

```
stmtseq(stmtseq(S))    → stmt(S).
stmtseq(stmtseq(S,Ss)) → stmt(S), stmtseq(Ss).
stmt(stmt(S))          → assign(S).
stmt(stmt(S))          → ifstmt(S).
assign(assign(Var,Exp)) → var(Var), [:=], expr(Exp).
ifstmt(if(C,T))        → [if], cond(C), [then], stmtseq(T).
ifstmt(if(C,T,E))      → [if], cond(C), [then], stmtseq(T), [else], stmtseq(E).
```

To specify the criterion that each `else` pair up with the closest previous unpaired `then`, we simply add the following preference clause:

```
ifstmt(if(C,if(C1,T),E)) --> Pf(ifstmt(if(C,if(C1,T,E))))
```

In general, if we have a sentential form with k parses, we would have $k-1$ arbiter clauses to specify the preferred parse.

We now briefly describe Preference Logic Programs [GJM94] and then show that PLGs can be translated into Preference Logic Programs.

3.2 Preference Logic Programming

Preference Logic Programs have two parts, a first order theory $\mathcal{T}$ and an arbiter $\mathcal{A}$. The first-order theory has three kinds of predicates:

1. *C-predicates* (core predicates) are defined using other core predicates through definite clauses.
2. *Opt-predicates* (optimization predicates) are defined using $\rightarrow$ clauses.
3. *O-predicates* are defined using $\leftarrow$ clauses and there is some goal in the body of a clause defining a O-predicate whose head is an Opt-predicate or an O-predicate.

The first-order part of a preference logic program is built up of the following kinds of clauses:

1. Core clauses of the form $H \leftarrow B_1, \ldots, B_n$, where each B_i is defined using other core clauses.
2. Optimization clauses of the following two forms
   - $H \rightarrow B_1, \ldots, B_m$. H is an Opt-predicate and each of the B_i could be a C-predicate, an Opt-predicate or an O-predicate. Note the direction of the implication in this formula. The intuition here is that the set of optimal solutions (*i.e.,* the set of binding for variables in the head of a $\rightarrow$ clause) is some subset of the feasible solutions (*i.e,* the set of bindings for the variables in the body).

- Definite clauses of the form $H \leftarrow B_1, \ldots, B_n$, where at least one B_i is an Opt-predicate or an O-predicate.

The first order theory $\mathcal{T}$ can therefore be divided into two parts, the definitions of the C-predicates that make up the core program $\mathcal{T}_C$ and the definitions of the Opt-predicates and the O-predicates that make up the optimization program $\mathcal{T}_O$.

The arbiter part of a preference logic program has the following form:

$$p(\bar{t}) \;\rightarrow\; \mathcal{P}_f(p(\bar{u}) \wedge \textstyle\bigwedge_i L_i) \qquad\qquad (i \geq 0)$$

where p is an Opt-predicate and each L_i is some atom. In all examples in this paper, each L_i is a constraint ($\geq, \leq, <, >$, *etc.*) over some domain. In essence this form of the arbiter states that the solution to the goal p in $\bar{u}$ is better than the solution in $\bar{t}$ because of $\bigwedge_i L_i$.

A preference logic program $\mathcal{P}$ can thus be characterized as a 3-tuple $\langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A} \rangle$, where $\mathcal{T}_C$ and $\mathcal{T}_O$ together form $\mathcal{T}$ and $\mathcal{A}$ is the arbiter.

For example, suppose we had the following preference logic program, which computes the list of edges along the shortest path from a source to a target node:

$$\begin{aligned}
path(X,Y,C,[e(X,Y)]) &\leftarrow edge(X,Y,C).\\
path(X,Y,C,[e(X,Z)|L1]) &\leftarrow edge(X,Z,C1),\ path(Z,Y,C2,L1),\ C = C1 + C2.\\
\\
sh_dist(X,Y,C,L) &\rightarrow path(X,Y,C,L).\\
sh_path(X,Y,L) &\leftarrow sh_dist(X,Y,C,L).\\
\\
sh_dist(X,Y,C1,L) &\rightarrow \mathcal{P}_f(sh_dist(X,Y,C2,L1) \wedge C2 < C1).
\end{aligned}$$

The first two definite clauses (and the clauses for the *edge* predicate) make up the *core program* $\mathcal{T}_C$. The third $\leftarrow$ clause and the $\rightarrow$ clause make up the *optimization program* $\mathcal{T}_O$. The only Opt-predicate is *sh_dist* and the only O-predicate is *sh_path*. The C-predicates are *path*, *edge* and $=$. [GJM94] describes the model theory of preference logic programs in detail and provides many other examples of preference logic programs.

3.3 From PLGs to Preference Logic Programs

The translation of PLGs to PLPs builds on the translation from DCTGs to CLPs. The translation can be broken into the following parts:

1. Nonterminals in the DCTG that do not have any preference rule attached to them nor depend upon any nonterminal that has a preference rule attached to it are translated in the usual way that DCTGs are translated into CLP programs. The resulting clauses form the core clauses in the target PLP program.

2. Nonterminals that have preference rules attached to them give rise to Opt-predicates (and their associated definitions) in the target PLP program.

3. Nonterminals that depend upon other nonterminals that have one ore more preference rules attached to them give rise to O-predicates in the target PLP program.

We briefly describe how nonterminals with preference rules are translated into PLP using the line-breaking example. Given the definition of `para` as before, we introduce a new predicate called `pref_para`, which stands for the preferred `para` (*i.e.*, the optimal paragraph) and has the following definition:

```
pref_para(node(pref_para,[Para],[badness(B)]),In,Out) →
         para(node(para, [Para],[badness(B)]),In,Out).
pref_para(node(pref_para,[Para],[badness(B)]),In,Out)→
 Pf(pref_para(node(pref_para, [Para1],[badness(B1)]),In,Out), B1 < B).
```

The occurrence of `para` on the r.h.s. of the core clauses resulting from the translation of the DCTG would have to be replaced by `pref_para`.

For the "dangling else" DCG, we similarly introduce a new predicate `pref_ifstmt`, for preferred `ifstmt`, with the following definition:

```
pref_ifstmt(S,In,Out) → ifstmt(S,In,Out).
pref_ifstmt(if(C,if(C1,T),E),In,Out) --> Pf(pref_ifstmt(if(C,if(C1,T,E)),In,Out)).
```

The occurrence of `ifstmt` on the r.h.s. of the core clauses resulting from the translation of the DCG would have to be replaced by `pref_ifstmt`.

4 Operational Semantics for Preference Logic Programs

We now present a top-down derivation scheme for computing the optimal answers. At the outset, it should be noted that we do not incur the expense of general theorem proving in modal logic because we are only interested in computing preferential consequences. The derivation scheme is an extension of SLD-resolution where some of the derivation paths get pruned due to the arbiter: the arbiter can thus be thought of as offering control advice to the SLD engine about which paths are better. Unlike the negation based approach of [GGZ91, Fag93], our derivation scheme performs optimization by explicit selection, *i.e.* it prunes paths that compute sub-optimal solutions.

Below we present an extension of SLD-resolution assuming that the core program consists of definite clauses without constraints; we subsequently describe how this scheme can be extended to the case where the core program is a constraint logic program.

Definition 1 *Let P be a definite clause program and G a goal. A* **partial SLD-tree** *for $P \cup \{G\}$, is a finite SLD-tree, not all of whose branches are successful or failed derivations of $P \cup \{G\}$.*

Because every edge in the SLD-tree is labeled by a substitution, we associate with each node in the SLD-tree a substitution which is the composition of the substitutions found on the path from the node to the root of the tree.

Definition 2 *Given two partial SLD-trees T_1 and T_2 for $P \cup \{G\}$, we define $T_1 \Rightarrow T_2$ to mean that T_2 is* **derived** *from T_1 by choosing a non-empty leaf $l = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ of T_1, A_m being the selected goal, and creating children of l of the form:*

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$$

for every clause $A \leftarrow B_1, \ldots, B_q$ in P such that θ is the most general unifier of A_m and A. The leaf l is said to be **expanded** *in T_1 to get T_2.*

We write $T_1 \Rightarrow^a T_2$ if we get T_2 from T_1 by simultaneously expanding all the expandable leaves in T_1.

Definition 3 *Let P be a definite clause program and G be a goal. A* **Tree** *SLD-derivation (***TSLD** *derivation) of $P \cup \{G\}$ is a finite or infinite sequence $T_0 = G, T_1, \ldots$ of partial SLD-trees for $P \cup \{G\}$ such that for all i, $T_i \Rightarrow^a T_{i+1}$.*

Definition 4 *Given a Preference Logic Program PLP= $\langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A} \rangle$, and a query G, a* **TSLD derivation for** *PLP$\cup\{G\}$ is an TSLD- derivation for $\mathcal{T}_C \wedge \mathcal{T}_O$. The $\rightarrow$ clauses are treated exactly as the $\leftarrow$ clauses.*

Definition 5 *Given a partial SLD-tree T for $P \cup \{G\}$, a node $n_1 = \leftarrow A_1, \ldots, A_j$ in T is said to be* **blocked** *if there exists another node $n_2 = \leftarrow B_1, \ldots, B_k$ and an internal node $n = \leftarrow D_1, \ldots, D_m, \ldots, D_n$, such that n_1 and n_2 are descendants of n, where D_m is $p(\overline{t})$ where p is an Opt-predicate, and is subject to an arbiter of the form:*

$$p(\overline{a}) \rightarrow \mathcal{P}_f(p(\overline{a_1}) \wedge \textstyle\bigwedge_i L_i)$$

In addition θ_1 and θ_2 are the substitutions associated with nodes n_1 and n_2 such that the following constraint is satisfiable:

$$\{p(\overline{t})\theta_1 = p(\overline{a}) \quad , \quad p(\overline{t})\theta_2 = p(\overline{a_1})\} \quad \cup \quad \textstyle\bigcup_i \{L_i\}$$

The substitution θ_2 is said to be **better** *than θ_1.*

A path in a partial SLD-tree that passes through a blocked node is said to be a pruned path. Given a preference logic program *PLP=* $\langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A} \rangle$, a Pruned TSLD-derivation is a TSLD-derivation in which, at each step, the leaf to be expanded is not a descendant of a blocked node. A tree occuring in a PTSLD derivation is said to be complete if all its paths are either successful, failed or pruned. A PTSLD-derivation $T_0, \ldots, T_s$ is *complete* if it ends in a complete tree. T_s is said to be the *result* of the complete PTSLD-derivation. [GJM94] also introduces the notion of a preferential consequence and the PTSLD-derivation scheme is sound but incomplete for computing answers with respect to this notion of consequence.

We now describe the extension of the derivation scheme to the case when the core program is a constraint logic program. The key difference between definite clause programs and constraint logic programs is that unification is replaced by a more general mechanism, namely constraint solving. Each node in the SLD-tree for CLP programs is characterized by a pair, namely a set of goals and a set of constraints. We briefly describe below how the derivation would proceed in the CLP domain by defining how successive trees are derived and how nodes can get blocked.

Definition 6 *Given a CLP program P, a goal G and two partial SLD-trees T_1 and T_2 for $P \cup \{G\}$, we define $T_1 \Rightarrow T_2$ to mean that T_2 is* **derived** *from T_1 by choosing a non-empty leaf $l = \langle\{A_1, \ldots, A_m, \ldots, A_k\}, \{C_j\}\rangle$ of T_1, A_m being the selected goal, and creating children of l of the form:*

$$\langle\{A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k\}, \{C_j\} \cup \{C\} \cup \{C'_i\}\rangle$$

for every clause $A \leftarrow C'_1, \ldots, C'_k, B_1, \ldots, B_q$ in P such that $\{C\}$ is the set of constraints generated by the equation $A_m = A$, and the C'_is in the body of the clause are constraints subject to the condition that $\{C_j\} \cup \{C\} \cup \{C'_i\}$ is solvable. The leaf l is said to be **expanded** *in T_1 to get T_2.*

Definition 7 *Given a partial SLD-tree T for $P \cup \{G\}$, a node $n_1 = \langle\{A_1, \ldots, A_j\}, \{C_{n_1}\}\rangle$, a node $n_2 = \langle\{B_1, \ldots, B_k\}, \{C_{n_2}\}\rangle$ and an internal node $n = \langle\{D_1, \ldots, D_m, \ldots, D_{n'}\}, \{C_n\}\rangle$, such that n_1 and n_2 are descendants of n, where D_m is $p(\overline{t})$ where p is an Opt-predicate, and is subject to an arbiter of the form:*

$$p(\overline{a}) \rightarrow \mathcal{P}_f(p(\overline{a_1}) \wedge \textstyle\bigwedge_i L_i)$$

In addition suppose the constraint

$$\{p(\overline{t}) = p(\overline{a})\} \cup \{p(\overline{t}) = p(\overline{a_1})\} \cup \textstyle\bigcup_i \{L_i\} \cup \{C_{n_1}\} \cup \{C_{n_2}\}$$

is satisfiable by the substitution η such that the projection (γ) of η to the variables in $\{C_{n_1}\}$ satisfies the constraint $\{p(\overline{t}) = p(\overline{a})\}$ and the projection to the variables in $\{C_{n_2}\}$ satisfies the constraint $\{p(\overline{t}) = p(\overline{a_1})\}$. We then update the constraint $\{C_{n_1}\}$ of node n_1 to $\{C_{n_1}\} \cup \{\neg\gamma\}$, where $\{\neg\gamma\}$ is a constraint that states that γ is not a solution. The solution γ is said to have been **blocked**, *a node in the tree is said to be blocked if all the solutions to the constraints of the node get blocked.*

In contrast with the previous definition for simple definite programs, each node in a SLD-tree in the CLP framework has a constraint associated with it which may be satisfiable in more than one way. Therefore each node in the SLD-tree in the CLP framework abstracts a set of solutions. The addition of a constraint $\{\neg\gamma\}$ blocks the solution γ. Note further that the nodes n_1 and n_2 in the definition need not be different nodes, *i.e.* one solution to the set of constraints may block another solution to the set of constraints.

The operational semantics presented here mimics the execution of the line-breaking algorithm in [KP81] for that example. The list of active modes in any tree in the derivation, corresponds to the active list in the line breaking algorithm in [KP81].

5 Incremental Computation

We outline a simple extension to the operational semantics to handle incremental optimal parsing. Incrementality is a very important component of interactive systems. The framework presented in this paper can serve as the basis of a WYSIWYG document editor only if optimal parsing were done incrementally; re-parsing from scratch each time a small change is made to the document is clearly infeasible. Our approach was also motivated by previous works on incremental attribute evaluation in language-based editors [Rep84], augmented parsers for supporting incrementality [GM80], and expanded querying power in constraint logic programming languages [MS89, vH90].

We now briefly describe the incremental algorithm to perform optimal parsing. The optimal parsing process can be viewed as a sequence of active lists where each active list is a list of partial parses and the next active list is obtained from the current one by expanding each partial parse. Let us assume that the operational semantics described in the previous section does a left-right parse of the input.

Input : A DCTG G with start symbol S, the current query $Q = S(Parse, In, [\,])$, the current PTSLD-derivation $\mathcal{T} = T_0, T_1, \ldots$ and a query $Q' = S(Parse1, In1, [\,])$ such that $In = x.y.z$ and $In1 = x.\overline{y}.z$, where . is the concatenation operator.

Output : The PTSLD-derivation for the query Q'.

Outline of incremental algorithm Given a PTSLD derivation $T_0, T_1, \ldots$, of trees, the active nodes in each tree represent partial parses. If the operational semantics, by its goal selection strategy enforces each parse to proceed left to right then the incremental algorithm can reuse a substantial portion of the previous computation. Each active node in an intermediate PTSLD derivation corresponds to a partial parse that has consumed some portion of the input. If the input list In changes to $In1$ both of which have the common prefix x, we can reuse the trees in the PTSLD derivation which have not parsed beyond x. So, we can start the optimal parse of $In1$ from the last tree in the PTSLD derivation for the input In such that none of the partial parses in it has reached a symbol in y. Since y has changed to $\overline{y}$, we need to construct the PTSLD derivation as long as the partial parses are parsing symbols in $\overline{y}$. Once a partial parse in a PTSLD tree on input $In1$ is in the region z, *i.e.*, has scanned past the change in the input, we can check for a matching condition to determine whether we can reuse portions of parses from the previous PTSLD derivation using the [GM80] matching condition.

The operational semantics will have to be modified to store pointers from the each position in the input list to an appropriate position in the list of trees making up the derivation tree. If a change is made starting at some position in the input list, we can recover the tree in the old PTSLD derivation from which we can start the current derivation using these pointers. These pointers serve the same purpose that the pointers in the threaded parse tree that [GM80] serve.

6 Conclusions and Status

We have presented an extension of DCTGs called Preference Logic Grammars, and shown their use for ambiguity resolution in logic grammars and for specifying optimal parsing problems. We believe that the use of preferential statements in logic grammars is concise, natural, and declarative. Although this extension was originally motivated by the extension to attribute grammars to specify document layout [BMW92], this paper shows that the ideas are applicable in a broader setting. Thus far we have considered only representative problems from the areas of document representation and programming language syntax. We still need to test our methodology on more problems, especially from the domain of natural language processing, where we believe the use of preferential statements for selection (as opposed to optimization) has a potential role.

In a related paper [GJM94], we have further elaborated on the paradigm of preference logic programming—its uses, declarative and operational semantics. Our proposed operational semantics has the potential to be efficiently implemented, although we have not yet done so at the time of writing this paper. In this regard, it may be noted that if the cost functions are well-behaved, one can show that the optimal parse can be obtained by constructing only a polynomial number of partial parses even if the underlying grammar is exponentially ambiguous. Finally, it should be noted that we have only sketched our ideas on incremental optimal parsing, and more work needs to be done in making them precise.

Acknowledgements

This research was supported by a grant from the XEROX Research Foundation. Surya Mantha would like to acknowledge the encouragement of Prof. Howard Blair and Prof. Anil Nerode during the development of the theory of preference logic.

References

[Abr84] H. Abramson. Definite Clause Translation Grammars. In *Proc. International Symposium on Logic Programming*, pages 233–241, Atlantic City, 1984. IEEE, Computer Society Press.

[AD89] H. Abramson and V. Dahl. *Logic Grammars.* Symbolic Computation. Springer-Verlag, 1989.

[AGT89] A. V. Aho, M. Ganapathi, and S. Tjiang. Code Generation using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[BMW92] A. Brown, S. Mantha, and T. Wakayama. The Declarative Semantics of Document Processing. In *Proc. of the First International Workshop on Principles of Document Processing*, 1992.

[Fag93] F. Fages. On the Semantics of Optimization Predicates in CLP Languages. In *Proc. of 13th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, 1993.

[GGZ91] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and Maximum Predicates in Logic Programming. In *Proc. Tenth ACM Symp. on Principles of Database Systems*, pages 154–163, 1991.

[GJM94] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference Logic Programming : Optimization as Inference. Technical Report 94-12, Dept. of Computer Science, SUNY at Buffalo, April 1994. Submitted for Publication.

[GM80] Carlo Ghezzi and Dino Mandrioli. Augmenting Parsers to Support Incrementality. *Journal of the ACM*, 27(3):564–579, July 1980.

[JL87] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987.

[JM] J. Jaffar and M. J. Maher. Constraint Logic Programming : A Survey. *To appear in the Journal of Logic Programming.*

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[KP81] D. E. Knuth and M. F. Plass. Breaking Paragraphs into Lines. *Software Practice and Experience*, 11:1119–1184, 1981.

[Mal] J. Maluszynski. Attribute Grammars and Logic Programs : A Comparison of Concepts. In H. Albas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, number 545 in LNCS, pages 330–357. Springer-Verlag.

[Man91] S. Mantha. *First-Order Preference Theories and their Applications.* PhD thesis, University of Utah, November 1991.

[MS89] M. J. Maher and P. J. Stuckey. Expanding Query Power in Constraint Logic Programming Languages. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 20–36, Cleveland, Ohio, USA, 1989.

[Rep84] T. W. Reps. *Generating Language-based Environments.* PhD thesis, Cornell University, Department of Computer Science, Ithaca, NY, 1984. ACM Distinguished Dissertation Series, MIT Press.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[vH90] P. van Hentenryck. Incremental constraint satisfaction in logic programming. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 189–202, Jerusalem, 1990. The MIT Press.