PARSING 2-D LANGUAGES WITH POSITIONAL GRAMMARS

Gennaro Costagliola and Shi-Kuo Chang

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
gencos@speedy.cs.pitt.edu
Phone: (412) 624-8836
FAX: (412) 624-8465

ABSTRACT

In this paper we will present a way to parse two-dimensional languages using LR parsing tables. To do this we describe two-dimensional (positional) grammars as a generalization of the context-free string grammars. The main idea behind this is to allow a traditional LR parser to choose the next symbol to parse from a two-dimensional space. Cases of ambiguity are analyzed and some ways to avoid them are presented. Finally, we construct a parser for the two-dimensional arithmetic expression language and implement it by using the tool Yacc.

INTRODUCTION

One of the latest approaches in parsing 2-D languages has been presented by Tomita in [37], where he introduces a 2-D Chomsky Normal Form grammar and constructs extensions to the two-dimensional case of Earley's and LR parsing algorithms.

In this paper, we present an extension of a context-free grammar by explicitly describing the positional relations between the elements (terminals and non-terminals) in the right hand-side of each production rule of the grammar. As these relations can be very general, the resulting grammar can be seen as a generalization of Tomita's 2-D Chomsky Normal Form grammar where only horizontal and vertical relations are allowed.

The resulting parser for such a positional grammar is constructed by simply adding a column to the LR parsing table. This column contains the position of the next symbol to be shifted, for each state. Unlikely from the 2-D LR parsing algorithms given in [37], our parser slightly modifies the original LR parsing algorithm, so that the tool Yacc can be easily used to construct a two-dimensional parser for a positional grammar.

Furthermore, we analyze cases of ambiguity, give some ways to avoid them and then present a general methodology to parse two-dimensional patterns applying it to the case of the two-dimensional arithmetic expressions.

Many other approaches have been proposed till now in high dimensional syntactic pattern representation and recognition. Each of them is based on the particular data structure used for representing the pictures: a string, an array, a tree, a graph, and a plex.

One of the first approaches is given by a traditional string grammar in which more general relations (HOR, VER, ABOVE, LEFT, etc.), other than concatenation, are allowed among primitives in the pattern [2, 8, 16]. Shaw, by attaching a "head" and a "tail" to each primitive, has used four binary operators for defining binary concatenation relations between primitives. A context-free string grammar is used to generate the resulting Picture Description Language (PDL) [16, 31].

Another interesting approach using a string grammar, has been given in [5] where each primitive has associated spatial attributes.

A simple two-dimensional generalization of string grammars is to extend grammars for one-dimensional strings to two-dimensional arrays [23, 28, 35, 38]. The primitives are the array elements and the relation between primitives is the two-dimensional concatenation.

Pfalz and Rosenberg have extended the concept of string grammar to grammars for labeled graphs called webs [16, 17, 26, 27, 29]. These grammars were originally suggested as a syntactical formalism for data structure useful in image analysis. An application of graph languages for describing scenes is of frequent occurrence in the literature dealing with image processing, whereas the use of graph grammars for pattern recognition is rare (for this purpose tree grammars are applied instead [3, 17, 18, 22, 30, 32]). Difficulties concerning building a syntax analyzer for graph grammars are causes of this situation. Recently, however, parsing methods for a particular kind of graph grammar have been proposed, and an efficient parsing, close to the parsing efficiency of tree languages, has been obtained [15, 21, 33].

Based on an idea in the work of Narasimhan [24], Feder [14] has formalized a "plex" grammar which generates languages with terminals having an arbitrary number of attaching points in order to connect to other primitives or subpatterns. The primitives of the plex grammar are called N-Attaching Point Entities (NAPEs). Plex structures defined by a plex grammar may be viewed as a hypergraph, with each NAPE corresponding to a hyperedge. Therefore this kind of plex grammar is a more general model than that of graph gram-

mar. Until recently, however, very little was known about the parsing method for plex grammars. Recently, a parsing method has been developed [25] to achieve more efficient parsing of plex grammars, by adapting Earley parsing algorithm, [13].

The paper is organized as follows. In Section 2 the positional grammar is defined, and some examples are given. In Section 3 the extension of the LR parser, named positional LR (pLR) parser, is presented along with a description of the pLR parsing tables and of the parsing algorithm. In Section 4 considerations of ambiguity are given along with the construction of a pLR parser for the arithmetic expression grammar. In Section 5 we present the general methodology for parsing 2-D languages generated by a positional grammar. The conclusions are in Section 6.

POSITIONAL GRAMMARS

The parser we are going to present recognizes pictorial languages generated by *positional* grammars.

Definition 2.1

A context-free *positional grammar* PG can be represented by a six-tuple (N, T, S, P, POS, PE) where:

N is a finite non-empty set of *non-terminal* symbols,
T is a finite non-empty set of *terminal* symbols,
$N \cap T = \emptyset$,
$S \in N$ is the *starting* symbol,
P is a finite set of *productions*
POS is a finite set of *positional relation* identifiers
$POS \cap (N \cup T) = \emptyset$,
PE is an *evaluation rule*

Each production in P has the following form:

$$A \rightarrow \alpha_1 REL_1 \alpha_2 REL_2 \cdots REL_{m-1} \alpha_m \qquad m \geq 1$$

where $A \in N$, each α_i is in $N \cup T$ and each REL_i is in POS. |

Each positional relation REL_i gives information about the relative position of α_{i+1} with respect to α_i. In the following, the words "positional grammar" will always refer to a context-free positional grammar.

While in a string grammar the only possible positional relation is the string concatenation, in a positional grammar other positional relations can be defined and then used for describing high dimensional languages. When parsing, this positional information will be useful for letting the scanner know where the next symbol to parse is.

Some simple examples of positional relations on a Cartesian plane:

String concatenation or adjacent horizontal concatenation

$AHOR = \{(p_1, p_2) : p_1$ and p_2 are pictures horizontally concatenated with alignment of their centroids$\}$

Adjacent vertical concatenation

$AVER = \{(p_1, p_2) : p_1$ and p_2 are pictures vertically concatenated with alignment of their centroids$\}$

Upper horizontal concatenation

$UHOR = \{(p_1, p_2) : p_1$ and p_2 are pictures horizontally concatenated with alignment of the centroid of p_1 and the up-most element of $p_2\}$

Horizontal concatenation

$HOR = \{(p_1, p_2) : p_1$ and p_2 are pictures and location$(p_1) = (x, y)$ and location'$(p_2) = (x', y')$ and the position (x', y') is *feasible* and $x' > x\}$

Vertical concatenation

$VER = \{(p_1, p_2) : p_1$ and p_2 are pictures and location$(p_1) = (x, y)$ and location'$(p_2) = (x', y')$ and the position (x', y') is *feasible* and $y' < y$ and $x' \leq x\}$

where a picture is a spatial arrangement of one or more symbols, location(p) is a function returning the position of a symbol of the picture p and a feasible location is a location that has not been made unfeasible by another symbol or by the side effect of an evaluation rule, as it will be seen in the following.

Definition 2.2

An *evaluation rule* PE is a function whose input is a string

$$p_1 REL_1 p_2 REL_2 \cdots REL_{m-1} p_m \qquad m \geq 1$$

where each p_i is a picture and each REL_i is a positional relation; its output is a picture whose elements $p_1, p_2, \ldots, p_m$ are disposed in the space such that

$$(p_i, p_{i+1}) \in REL_i \qquad 1 \leq i \leq m-1.$$

The evaluation of the positional relations is meant to be sequential from left to right. As side effects can be generated for any evaluation, an *evaluation rule* is *simple* if no side effects are involved. |

A possible side effect of the evaluation of a relation is to make certain positions in the space unfeasible. As the evaluation is sequential, each evaluation inherites the side effects generated by the previous evaluations.

Some examples of applications of the simple evaluation rule follow:

PE("a . b . c . d") = $a\ b\ c\ d$

PE("a *VER* b *HOR* c") =
$$\begin{matrix} & a & \\ b & & c \end{matrix}$$

PE("a *AVER* b") =
$$\begin{matrix} a \\ b \end{matrix}$$

where the positional relations '.', *VER*, *HOR* and *AVER* are defined as above.

The following definitions are understood to be with respect to a particular positional grammar G.

We write $\Pi \Rightarrow \Sigma$ if there exist Δ, Γ, A, η such that $\Pi = \Gamma A \Delta$, $A \rightarrow \eta$ is a production and $\Sigma = \Gamma \eta \Delta$.

We write $\Pi \Rightarrow^* \Sigma$ (Σ is derived from Π) if there exist strings $\Pi_0, \Pi_1 \cdots \Pi_m$ ($m \geq 0$) such that

$$\Pi = \Pi_0 \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_m = \Sigma$$

The sequence $\Pi_0, ..., \Pi_m$ is called a derivation of Σ from Π. A *positional sentential form* is a string Π such that $S \Rightarrow^* \Pi$. A *positional sentence* is a positional sentential form with only terminal symbols. A *pictorial form* is the evaluation of a positional sentential form. A *picture* is a pictorial form with only terminal symbols. The *pictorial language defined by a positional grammar L(G)* is the set of its pictures.

Some examples of positional grammars:

2.1) The following grammar generates the strings of the form $a \cdots ab \cdots b$ with equal number of a's and b's.

N = {S}
T = {a, b}
POS= {.}
PE is the simple evaluation rule
P = {
S := a . S . b | a . b
}

The positional operator . is defined as above. A positional sentence of this grammar is: a . a . a . b . b . b and the corresponding picture is: aaabbb.

This example shows that every context-free string language can be represented by a positional grammar.

2.2) The following grammar generates an upper-right corner with variable length of the edges.

N = {Corner, HLine, VLine}
T = {*dot*}
S = Corner
POS= {*UHOR*, *AHOR*, *AVER*}
PE is the simple evaluation rule
P = {
Corner := HLine *UHOR* VLine
HLine := HLine *AHOR dot* | *dot*
VLine := VLine *AVER dot* | *dot*
}

where *UHOR*, *AHOR* and *AVER* are defined as above. A positional sentence of this grammar is:

dot AHOR dot AHOR dot AHOR dot UHOR dot AVER dot AVER dot AVER dot

Replacing *dot* with the character '.', the corresponding picture is:

2.3) The following grammar generates two-dimensional arithmetic expressions using the binary operations addition and division:

N = {E, T, F}
S = E
T = {+, *hbar*, (,), *id*}
POS= {*HOR*, *VER*}
PE is the evaluation rule defined below
P = {
E := E *HOR* + *HOR* T | T
T := T *VER hbar VER* F | F
F := (*HOR* E *HOR*) | *id*
}

The evaluation rule is so defined (see Figure 2.1):

PE(p_1 *HOR* p_2):
The evaluation of *HOR* will give coordinates (x, y) to location(p_1) and (x', y') to location(p_2) such that (p_1, p_2) $\in$ *HOR*. Moreover it will make unfeasible each position belonging to any of the following sets:
$\{(x, y_1) : y \leq y_1 \leq m\}$
$\{(x_1, y_2) : x < x_1 < x' \text{ and } 0 \leq y_2 \leq m\}$
$\{(x', y_3) : y' \leq y_3 \leq m\}$
where $m \geq 1$ is an upper bound on the y-coordinate in the two-dimensional space.

PE(p_1 *VER* p_2):
The evaluation of *VER* will give coordinates (x, y) to location(p_1) and (x', y') to location(p_2) such that (p_1, p_2) $\in$ *VER*. Moreover it will make unfeasible each position belonging to any of the following sets:
$\{(x_1, y) : 0 \leq x_1 \leq x\}$
$\{(x_2, y_1) : 0 \leq x_2 \leq x \text{ and } y' < y_1 < y\}$
$\{(x_3, y') : 0 \leq x_3 \leq x'\}$

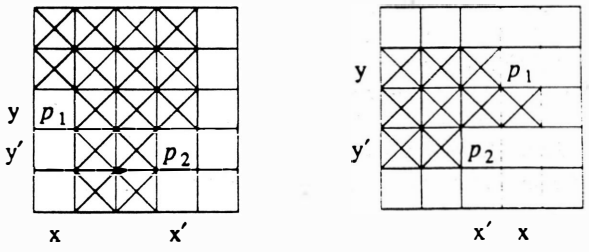


Figure 2.1. {p_1 *HOR* p_2} and {p_1 *VER* p_2}

A positional sentence of this grammar is:

id HOR + HOR (HOR id HOR + HOR id HOR) VER hbar VER id HOR + HOR id

Replacing *hbar* with an horizontal bar, according to the definitions of *HOR*, *VER* and PE, there are many possible pictures corresponding to the evaluation of this positional sentence, but all of them can be mapped into the following one:

$$id + \frac{(id + id)}{id} + id$$

that is still a picture of this language.

POSITIONAL LR PARSERS

Positional LR parsers (pLR parsers) are nothing else but a generalization of the LR parsers. The model of a pLR parser is given by:

1) Input
2) Positional operators
3) pLR Parsing Table
4) pLR Parsing Program
5) Stack
6) Output

as shown in Figure 3.1.

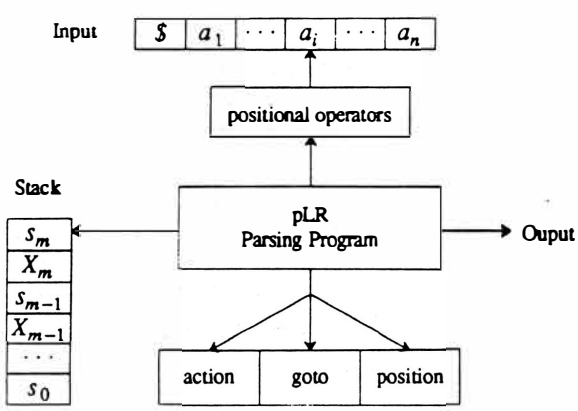


Figure 3.1. The model of a pLR Parser

The input

The input to a pLR parser is a spatial arrangement of tokens, or, in other words, a symbolic picture where each symbol is a token. Such an input is represented by an array w (the input tape) where each token is stored, a list Q of couples (*pos*, *i*) where *pos* is the spatial position of the token w[*i*], and a starting index that points to the first token to parse. The association between a position and a token allows us to reach a token in w each time its spatial position has been given and viceversa. The input tape is, then, no longer required to be accessed sequentially but rather, according to the positional requirements given by the parser.

In this context, the definition of the sequential end-of-string marker must be extented. In fact, the end-of-string marker hides an operational aspect: when parsed, it signals that no symbols to parse are left. While in a sequential scanning nothing must be done other than recognizing the '$' character, in a non-sequential scanning such operational aspect must be made explicit. Before returning an end-of-input symbol, the scanner has to check whether all the symbols have been parsed.

In a pLR parser, the end-of-input marking is implemented by storing the symbol '$' in location 0 of the input tape, and defining the *end-of-input operator* ANY as a function whose return value is 0 if all the symbols in the input tape have been parsed and 'error' otherwise.

The positional operators

For each positional relation we define a positional operator with the same name. Such an operator is a function that takes in input the index in the tape of the last token parsed, calculates a new position and then returns the index of the next token to parse, by consulting the list Q.

Definition 3.1

Given a positional grammar PG = (N, T, S, P, POS, PE) and a relation *REL* ∈ POS, then for all α, β ∈ N ∪ T such that "α *REL* β" occurs on the right hand-side of a production rule in PG, the *corresponding positional operator* REL is defined as follows:

REL(i) = j iff i is the index in w of 'a', the last token parsed to reduce α, and j is the index in w of 'b', the first token to parse to reduce β. |

Examples:

3.1) In the grammar of Example 2.2, the corresponding operators for POS can be defined as follows:

UHOR(i) = AHOR(i) = j iff location(w[i]) = (x, y) and location(w[j]) = (x+δ, y).

AVER(i) = j iff location(w[i]) = (x, y) and location(w[j]) = (x, y-δ).
where δ is the distance between each couple of dots.

3.2) For the arithmetic expression grammar the operators HOR and VER can be defined as follow:

HOR(i) = j iff location(w[j]) is the highest spatial position in the first non-empty column on the right of location(w[i]).

VER(i) = j iff location(w[j]) is the spatial position on the left of location(w[i]) such that it is the leftmost position in the first non-empty row below location(w[i]).

The Positional LR Parsing Table

Besides the "action" and "goto" columns of an LR parsing table, the pLR parsing table contains an additional column called "position". The positional operators SP, ANY and the names of the positional operators are the elements of this new column. SP returns the starting index given in input with the picture and ANY is the operator defined above. All the names

in the column "position" can be considered as pointers to the code implementing the operators.

As the construction of the "position" column does not affect the other entries of the original LR parsing table, we can use the traditional three techniques (with some variations) for having Simple pLR, canonical pLR and LookAhead pLR parsers.

A pLR(0) item of a positional grammar PG is a production of PG with a dot at some position of the right side. A dot, however, can never be between a positional operator identifier and either a terminal or a non terminal, in this order. Thus, a production A → SP X REL_1 Y REL_2 Z yields the four items:

A → .SP X REL_1 Y REL_2 Z
A → SP X .REL_1 Y REL_2 Z
A → SP X REL_1 Y .REL_2 Z
A → SP X REL_1 Y REL_2 Z .

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the first item above indicates that we hope next to see a pattern derivable from XYZ starting from position SP. The second item indicates that we have just seen on the input a pattern derivable from X and that we hope next to see a pattern derivable from YZ starting from the position specified by the operator associated to REL_1.

If PG is a grammar with starting symbol S, then PG', the augmented positional grammar for PG, is PG with a new starting symbol S' and production S' := SP S.

Example 3.3

Let us consider the following positional grammar generating an horizontal concatenation of a block of squares, an arrow and another block of squares

(1) S := B1 *HOR* => *HOR* B2
(2) B1 := C *HOR* C
(3) C := square *VER* square
(4) B2 := R *VER* R
(5) R := square *HOR* square

Here the definition of PE is as in Example 2.3. The canonical collection of sets of pLR(0) items for this grammar follows next, along with the position values. The goto function for this set of items is shown as the transition diagram of a deterministic finite automaton in Figure 3.2 and the resulting Positional LR parsing table is given in Figure 3.3.

I_0: S' := .SP S — position[0] = {SP}
S := .B1 HOR => HOR B2
B1 := .C HOR C
C := .square VER square

I_1: S' := SP S. — position[1] = {ANY}

I_2: S := B1 .HOR => HOR B2 — position[2] = {HOR}

I_3: B1 := C .HOR C — position[3] = {HOR}
C := .square VER square

I_4: C := square .VER square — position[4] = {VER}

I_5: S := B1 HOR => .HOR B2 — position[5] = {HOR}
B2 := .R VER R
R := .square HOR square

I_6: B1 := C HOR C . — position[6] = {HOR}

I_7: C := square VER square . — position[7] = {HOR}

I_8: S := B1 HOR => HOR B2 . — position[8] = {ANY}

I_9: B2 := R .VER R — position[9] = {VER}
R := .square HOR square

I_{10}: R := square .HOR square — position[10] = {HOR}

I_{11}: R := square HOR square . — position[11] = {VER,ANY}

I_{12}: B2 := R VER R . — position[12] = {ANY}

Note that in the construction of each closure, the positional operators HOR and VER are ignored by the dot. This information is instead caught by the position array.

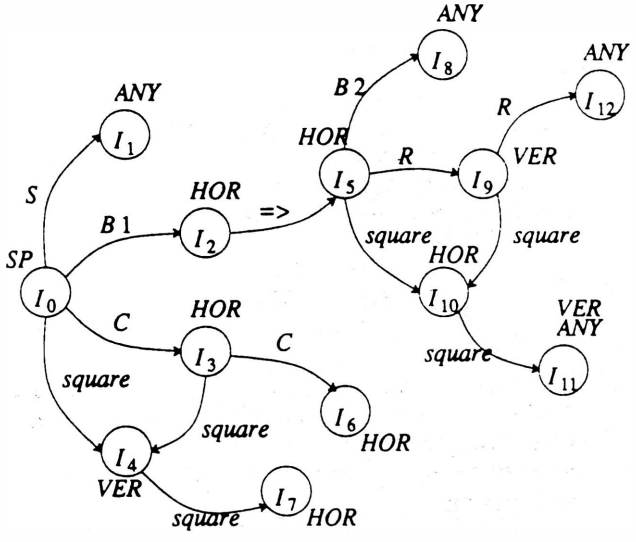


Figure 3.2. Transition diagram

state	action			goto					position
	square	=>	$	S	B1	B2	C	R	
0	s4			1	2		3		SP
1			acc						ANY
2		s5							HOR
3	s4						6		HOR
4	s7								VER
5	s10					8		9	HOR
6		r2							HOR
7	r3	r3							HOR
8			r1						ANY
9	s10							12	VER
10	s11								HOR
11	r5		r5						ANY VER
12			r4						ANY

Figure 3.3. A Simple pLR parsing table

Details on the algorithm for the construction of a Positional LR parsing table can be found in [9, 10].

The Positional LR Parsing Algorithm

The pLR algorithm is a simple extension of Algorithm 4.7 in [1]; the only differences regard the form of the input and the setting of the pointer to the next symbol.

The input is now given by a picture p represented by an array of tokens w, a starting index in w, and a list Q of couples (*pos*, *i*); the specification of a set of positional operators, and the pLR parsing table with functions "action", "goto" and "position" for a positional grammar PG.

Each time the pLr parser reaches a state in the recognition of the pattern, the next symbol to parse is determined by using the positional operator associated to that state. As in LR parsing, a same symbol cannot be considered more than once.

Details on the Positional LR parsing algorithm can be found in [9, 10].

Examples

3.4) Figure 3.4 shows the parsing action, goto and position of a canonical pLR parsing table for the following linear positional grammar for the vertical concatenation of two strings both of the type "c · · · cd".

(1) S := C *VER* C
(2) C := c *AHOR* C
(3) C := d

where the evaluation rule is simple when applyed to *AHOR* and defined as in Example 2.3 when applied to *VER*. Using the parsing table in Figure 3.4 and applying the pLR parsing algorithm, it can be verified that the following picture

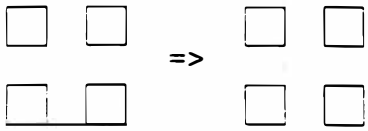
ccccccccccd
ccccd

is in the described language.

state	action			goto		position
	c	d	$	S	C	
0	s3	s4		1	2	SP
1			acc			ANY
2	s6	s7			5	VER
3	s3	s4			8	AHOR
4	r3	r3				VER
5			r1			ANY
6	s6	s7			9	AHOR
7			r3			ANY
8	r2	r2				VER
9			r2			ANY

Figure 3.4. A canonical pLR parsing table

3.5) Given the grammar in Example 3.3, using the parsing table in Figure 3.3 and applying the pLR parsing algorithm, it can be verified that the following picture



is accepted. In particular, note that the parser drives the scanning of the input such that the first block is visited by columns, and the second block by rows, according to the productions of the grammar. All the other ways of scanning this input are not taken into consideration.

AMBIGUITY CONSIDERATIONS

In this Section we will show that conflicts in positions can lead to conflicts in the "action" part of the parsing table even if it has no multiple entries.

In Section 2 we gave a two-dimensional version of the grammar given in [1] for arithmetic expressions. We will show now that this grammar is not pLR(1) from the fact that it has conflicts regarding the position of the next symbol. Let us consider the following pictorial form:

$$\frac{T}{id} + id$$

assuming that T has already been reduced.

After reducing T, the parser has to decide whether to choose '*hbar*' in vertical reading, or '+' in horizontal reading. Both the alternatives are valid: if '*hbar*' is chosen, then the parser has to shift, otherwise it has to reduce. One possibility for avoiding this conflict is to assign priority to each positional operator. In this example we could decide that the vertical reading has always higher priority than the horizontal one. This would respect the priority between '*hbar*' and '+' implicitly given in the grammar. But, if this other example is considered

$$\frac{(T + id)}{id} + id$$

the priority resolution will fail. In fact, in this case, after reading T, we want to move horizontally because of the parenthesis, and not vertically.

Another possibility for avoiding this conflict is to give a "smart" representation of the two-dimensional pattern deriving it from techniques of image analysis like dominancy [4, 12]. Last but not least, we can construct an equivalent pLR(1) grammar as it is normally done for solving conflicts in LR parsers. Following these ideas, the pLR(1) grammar for the arithmetic expressions has been constructed:

(0) E' := SP E
(1) E := E *HOR* + *HOR* T

(2) E := T
(3) T := T' *VER* F
(4) T := F
(5) F := (*HOR* E *HOR*)
(6) F := *id*
(7) T' := T' *VER* F'
(8) T' := F'
(9) F' := ⌊ *HOR* E *HOR* ⌋
(10) F' := *id*

Figure 4.1 shows the resulting pLR(1) parsing table for this grammar.

Note that the terminals *id*, (, and) have been duplicated as well as the non-terminals T and F. Moreover, rules (3), (4), (5) and (6) have been duplicated in rules (7), (8), (9) and (10). The new grammar, then, has a particular section dedicated to the generation of the numerator of any division.

During the recognition, this allows us to decide whether the expression to be parsed is the numerator of a division or not. In particular, the new terminals ⌊ and ⌋ mark the beginning and the end of any complex numerator, respectively, and the terminal *id* is the simple numerator.

state	action							goto						position
	$	*id*	+	)	(	*id*	⌋	⌊	E'	E	T	F	T'	F'	
0		s7			s5	s6		s8		1	2	4	3	9	SP
1	acc		s10												(HOR (ANY
2	r2		r2	r2			r2								HOR
3		s7			s5	s6		s8				12		13	VER
4	r4		r4	r4			r4								HOR
5		s7			s5	s6		s8		14	2	4	3	9	HOR
6	r6		r6	r6			r6								HOR
7		r10			r10	r10		r10							VER
8		s7			s5	s6		s8		15	2	4	3		HOR
9		r8			r8	r8		r8							VER
10		s7			s5	s6		s8			11	4	3	9	HOR
11	r1		r1	r1			r1								HOR
12	r3		r3	r3			r3								HOR
13			r7	r7			r7								HOR
14			s10	s16											HOR
15			s10				s17								HOR
16	r5		r5	r5			r5								HOR
17		r9			r9	r9		r9							VER

Figure 4.1. pLR parsing table for arithmetic expressions

A trace for the acceptance of the following patterns can be easily constructed

$$\begin{matrix} \lfloor id + id \rfloor \\ id \end{matrix} + id \qquad \begin{matrix} id \\ id + id \\ id \end{matrix}$$

AN IMPLEMENTATION

The general methodology to parse pLR languages is the following:

I. Define a general data structure to represent the two-dimensional symbolic pictures.

II. Define the positional relations and operators meant to relate objects in the patterns, and construct the pLR positional grammar, if possible, to describe the language.

III. Convert the general data structure into the input to the parser as defined in Section 3.

IV. Construct the parser.

Point I requires a general data structure to represent the original symbolic picture input. This can be a matrix of symbols, or an iconic index, i. e., an analogous linear representation based on the projections of the symbols: the 2-D string as defined in [6], or, for high dimensional symbolic patterns, the Gen_string, [11]. As the whole parsing model presented is extensible to the n-D case (n >2) just considering positional relations and operators for the n-dimensional space, we will make use of the Gen_string iconic index. The characteristics of it and the algorithms to derive it from a high dimensional pattern are given in [11]. In the proposed implementation, each element of the Gen_string is a token. A lexical analyzer to construct such a Gen_string can be obtained by using the same actions described above, but allowing the elements of the general data structure (another Gen_string) to be elementary items or pixels.

Point II requires the construction of the pLR linear positional grammar along with the positional operators.

Point III requires routines for the conversion of the general data structure into an array of tokens w, a starting index in w, SP, and an association list Q of positions and tokens. In particular the list Q must be implemented such that the positional operators can be executed efficiently.

Finally, Point IV requires the construction of the parser. As a result of Theorem 7.1 in[9], this can be done by translating the positional LR grammar into an LR grammar with actions and then by using the tool Yacc, [20].

As an example of the construction given in that Theorem, let us consider the the positional LR grammar for the arithmetic expressions. The resulting LR context free grammar with actions is:

(1) E := E + {HOR()} T
(2) E := T
(3) T := T' F
(4) T := F
(5) F := ({HOR()} E) {HOR()}
(6) F := *id* {HOR()}
(7) *T'* := *T' F'*
(8) *T'* := *F'*
(9) *F'* := ⌊ {HOR()} E ⌋ {VER()}
(10) *F'* := *id* {VER()}

An implementation by Yacc for this grammar, using the Gen_string representation, has been developed at the Department of Computer Science of the University of Pittsburgh. The implementation consists of the following:

The function *get_gs()*: the Gen_string representing a two-dimensional arithmetic expression is stored in a global data structure "gs". The Gen_string can be taken from a database or derived from the original pattern.

The function *gs_ir()*: the Gen_string is converted into an internal representation (data structure "spg", and others).

The functions *read_hor()* and *read_ver()*: the spatial operators HOR and VER are implemented, respectively.

The *yacc specifications* for the grammar: the functions read_hor() and read_ver() are inserted in the rules as actions. Both of them update a global variable "current" used by the function *yylex()* to select the next token to be parsed.

In the following, the results of the execution of such specifications are given. Note that the array "spg" represents the set of tokens occurring in the expression while the values of "current" give the order in which the tokens are parsed. For each token spg[i], the (x, y) coordinates are also given (the list Q). In this implementation x represents the column index in left-right progression, and y the row index in top-down progression.

Case 1

get_gs1: the input Gen_string is equivalent to

$$\frac{(99 + 501)}{6} * \frac{10}{2}$$

spg[0] = "\0"		
spg[1] = "("	x = 1	y = 1
spg[2] = "99"	x = 2	y = 1
spg[3] = "+"	x = 3	y = 1
spg[4] = "6"	x = 3	y = 2
spg[5] = "501"	x = 4	y = 1
spg[6] = ")"	x = 5	y = 1
spg[7] = "*"	x = 6	y = 2
spg[8] = "10"	x = 7	y = 1
spg[9] = "2"	x = 7	y = 2

current = 1 2 3 5 6 4 7 8 9 0 ... the result is -> 500

Case 2

get_gs2: the input Gen_string is equivalent to

$$\frac{8}{\left(\frac{4}{2}\right)} - 2$$

spg[0] = "\0"		
spg[1] = "("	x = 1	y = 2
spg[2] = "8"	x = 2	y = 1
spg[3] = "4"	x = 2	y = 2
spg[4] = "2"	x = 2	y = 3
spg[5] = ")"	x = 3	y = 2
spg[6] = "-"	x = 4	y = 2
spg[7] = "2"	x = 5	y = 2

current = 2 1 3 4 5 6 7 0 ... the result is -> 2

CONCLUSIONS

In this paper we constructed a parser for a subclass of symbolic pictorial languages. We showed that this class contains the context-free string languages and that a complex language like the two-dimensional arithmetic expression language can be parsed by the proposed model. We also showed that this class has a real nice property: the possibility to be parsed in a very simple way by using an existing tool.

At the moment we are investigating the extension of universal parsers like Earley's ([13]) and Tomita's ([36]) algorithms by applying the same technique used for extending the LR parser. Moreover we are considering applications of the model proposed to graphics and to visual languages ([7, 12, 19, 34]).

In the future we intend to extend the subclass of pictorial languages parseable by constructing more powerful parsers. A first approach regards the extension of the concept of symbol to an N-Attaching Point Entity as defined in [14]. A second approach regards instead the possibility to have more than one positional relation between two symbols. In this way a symbol can be connected to non-adjacent symbols, too.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques, and tools*, Addison Wesley, 1985.

[2] H.G. Barrow and J.R. Popplestone, "Relational Descriptions in picture processing," *Machine Intelligence*, vol. 6, pp. 377-396, 1971.

[3] N.S. Chang and K.S. Fu, "Parallel Parsing of Tree Languages for Syntactic Pattern Recognition," *Pattern Recognition*, vol. 11, no. 3, pp. 213-222, 1979.

[4] S.-K. Chang, "A Method for the Structural Analysis of Two Dimensional Mathematical Expressions," *Information Sciences*, vol. 2, pp. 253-272, 1970.

[5] S.-K. Chang, "Picture Processing Grammar and its Applications," *Information Sciences*, vol. 3, pp. 121-148, 1971.

[6] S.-K. Chang, Q.Y. Shi, and C.W. Yan, "Iconic Indexing by 2-D strings," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 4, pp. 475-484, July 1984.

[7] S.-K. Chang, M.J. Tauber, B. Yu, and J.S. Yu, "A Visual Language Compiler," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 506-525, 1989.

[8] M.B. Clowes, "Pictorial Relationships - A Syntactic Approach," *Machine Intelligence*, vol. 4, Amer. Elsevier, New York, 1969.

[9] G. Costagliola and S.-K. Chang, "Parsing Linear Pictorial Languages by Syntax-Directed Scanning," *submitted to JACM*.

[10] G. Costagliola and S.-K. Chang, "DR PARSERS: a generalization of LR parsers," *Proc. of 1990 IEEE Workshop on Visual Languages*, pp. 174-180, Skokie, Illinois, USA, October 4-6.

[11] G. Costagliola, G. Tortora, and T. Arndt, "A Unifying Approach to Iconic Indexing for 2-D and 3-D Scenes," *to appear in IEEE Transactions on Knowledge and Data Engineering*.

[12] C. Crimi, A. Guercio, G. Pacini, G. Tortora, and M. Tucci, "Automating Visual Language Generation," *IEEE Transactions on Software Engineering*, vol. 16 , no. 10, pp. 1122-1135, October 1990.

[13] J. Earley, "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, vol. 13, pp. 94-102, 1970.

[14] J. Feder, "Plex Languages," *Information Sciences*, vol. 3, pp. 225-241, 1971.

[15] M. Flasinski, "Characteristics of edNLC-Graph Grammar for Syntactic Pattern Recognition," *Computer Vision Graphics and Image Processing*, vol. 47, pp. 1-21, 1989.

[16] K.S. Fu, *Syntactic Methods in Pattern Recognition*, Academic Press, New York and London, 1974.

[17] K.S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice Hall, Inc. Englewood Cliffs, N.J., 1982.

[18] K.S. Fu and B.K. Bhargava, "Tree Systems for Syntactic Pattern Recognition," *IEEE Trans. Comput.*, vol. C-22 (12), pp. 1089-1099, 1973.

[19] E.J. Golin and S.P. Reiss, "The Specification of Visual Language Syntax," *Proc. of 1989 IEEE Workshop on Visual Languages*, pp. 105-110, Rome/Italy, October 4-6.

[20] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler," *tech. rep.*, Bell Laboratories, 1974.

[21] C.Y. Li, T. Kawashima, T. Yamamoto, and Y. Aoki, "Attribute Expansive Graph Grammar for Pattern Description and its Problem-reduction Based Processing," *Trans. IEICE*, vol. E-71 (4), pp. 431-440, Japan, 1988.

[22] S.Y. Lu and K.S. Fu, "Error-correcting Tree Automata for Syntactic Pattern Recognition," *IEEE Trans. Comput.*, vol. C-27, pp. 1040-1053, 1978.

[23] D.L. Milgram and A. Rosenfeld, "Array Automata and Array Grammars," *Information Processing* , vol. 71, pp. 69-74, North-Holland Publ., Amsterdam, 1972.

[24] R. Narasimhan, "Syntax-directed Interpretation of Classes of Pictures," *Comm. ACM*, vol. 9, pp. 166-173, 1966.

[25] K. J. Peng, T. Yamamoto, and Y. Aoki, "A New Parsing Scheme for Plex Grammars," *Pattern Recognition*, vol. 23 , no. 3/4, pp. 393-402, 1990.

[26] J. L. Pfaltz, "Web Grammars and Picture Description," *Comput. Graphics Image Processing*, vol. 1, pp. 193-220, 1972.

[27] J. L. Pfaltz and A. Rosenfeld, "Web Grammars," *Proc. of First Int. Joint Conf. Artif. Intell.*, pp. 609-619, Washington, DC, May 1969.

[28] A. Rosenfeld, *Picture Languages: Formal Models for Picture Recognition*, Academic Press, New York, San Francisco and London, 1979.

[29] A. Rosenfeld and D. L. Milgram, "Web Automata and Web Grammars," *Machine Intelligence*, vol. 7, pp. 307-324, 1972.

[30] W.C. Rounds, "Context Free Grammars on Trees," *Proc. of 10th Symp. Switching and Automata Theory*, p. 143, 1969.

[31] A.C. Shaw, "A Formal Picture Description Scheme as a Basic for Picture Processing Systems," *Information and Control*, vol. 14, pp. 9-52, 1969.

[32] Q.Y. Shi and K.S. Fu, "Efficient and Error-correcting Parsing of (attributed and stochastic) Tree Grammars," *Information Sciences*, vol. 26, pp. 159-188, 1982.

[33] Q. Y. Shi and K.S. Fu, "Parsing and Translation of Attributed Expansive Graph Languages for Scene Analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, pp. 472-485, 1983.

[34] N.C. Shu, *Visual Programming*, Van Nostrand Reinhold Company, 1988.

[35] G. Siromoney, R. Siromoney, and K. Krithivasan, "Array Grammars and Kolam," *Comput. Graphics and Image Processing*, vol. 3, pp. 63-82, 1974.

[36] M. Tomita, *Efficient Parsing for Natural Languages*, Kluwer Academic Publishers, Boston, MA, 1985.

[37] M. Tomita, "Parsing 2-Dimensional Languages," *Proceedings of the International Workshop on Parsing Technologies*, pp. 414-424, Pittsburgh, PA. Carnegie Mellon, 28-31 August 1989.

[38] P.S.P. Wang, "Recognition of Two-Dimensional Patterns," *Proc. Assoc. Comput. Mach. Nat. Conf.*, pp. 484-489, 1977.