Characterizing Unambiguous Precedence Systems in Expressions without Superfluous Parentheses

Wafik Boulos Lotfallah*

November 29, 2007

Abstract

When infix notation is used, parentheses are sometimes omitted according to a precedence relation between the operators as well as a classification of (binary) operators as being left or right associative. We analyze these concepts by first giving a definition of a general precedence system, that declares the superfluous parenthesis pairs for any given expression. We give a characterization of unambiguity in this general setting, and study the complexity of parsing expressions without superfluous parentheses. Also, we study the two notions of maximal unambiguous and complete precedence systems and give a characterization for each one of these notions. Finally, we show that complete precedence systems can be equivalently described by a chain of left associative and right associative classes of operators, with some extra restrictions on the relative positions and the associativity of unary operators.

Keywords: formal languages, superfluous parentheses, precedence order, infix notation, unambiguous grammars, parsing

AMS 2000 Mathematics Subject Classification codes: 03D05, 68Q42, 68Q45

Computing Classification System codes: F.4.2, F.4.3, G.2.2

1 Introduction

The use of parentheses in formal expressions has been studied from many points of view. First, the Polish logician Lukasiewicz showed in [5] that the use of prefix (or postfix) notation eliminates the role of parentheses altogether. Prefix notation was thus called Polish notation. Also, postfix notation is called reverse Polish notation and is widely used in compiler design (see e.g. [7]).

However, it seems that, in spite of that advantage, Polish notation is not compatible with the way formal expressions are perceived by humans. E.g. try to understand what is meant by the identity:

\[
= - \uparrow a 2 \uparrow b 2 \ast - a b + a b ,
\]

which can be transformed by the infix notation to just the readable and famous identity:

\[
(((a \uparrow 2) - (b \uparrow 2)) = ((a - b) \ast (a + b))) ,
\]

---

*Department of Mathematics, University of Wisconsin-Madison, USA, E-mail Address: lotfalla@math.wisc.edu, Department of Mathematics, The German University in Cairo, E-mail Address: wafik.lotfallah@guc.edu.eg
where parentheses are needed to resolve the syntax of the expression. The importance of infix notation pushed some research in developing faster algorithms to deal with parentheses, e.g. finding a matching parenthesis or parsing an arithmetic expression.

However, the above expression has many superfluous parentheses and can be further simplified to:

\[ a \uparrow 2 - b \uparrow 2 = (a - b) * (a + b) , \]

where we can use some natural precedence order on the set of operators (including the equality symbol) to recover the missing parentheses. The advantage of using such simplified expressions is that they are shorter and easier to perceive by humans.

The precedence order, as well as the left and right associativity of (binary) operators, has already been used in compiler design when parsing expressions in infix notation, e.g. transforming these to the reverse Polish notation (see [6]). Also, an algorithm for the reverse transformation to infix notation in the case of arithmetic expressions can be found in [3] and [4], and [1] gave a general schema for designing a context free grammar that distinguishes superfluous parentheses.

However, to avoid ambiguity, all the results so far have assumed that all operators are stratified in precedence levels, where operators on each level are either left- or right associative.

In this paper we study the concept of superfluous parentheses in a more general setting. Instead of making the above assumption, we allow arbitrary context-free declarations of superfluous parentheses, and characterize the unambiguity in this setting.

In particular, after giving the necessary definitions and background in Section 2,

1. We start Section 3 by giving a definition of a precedence system, which declares in the most general way for each matching parenthesis pair in an expression whether it is superfluous. We also define the notion of an unambiguous precedence system, and give a characterization of such systems (Theorem 3.4). It is interesting to see that the nontrivial direction of that theorem can be stated as an abstract theorem in discrete mathematics (Theorem 3.6). We also study the parsing complexity for general precedence systems (Proposition 3.8 with the appendix in Section 7).

2. In Section 4, we define the notion of a maximal unambiguous precedence system, which is an unambiguous system with no proper unambiguous extensions. We then give a characterization of such systems in Theorem 4.3.

3. In Section 5, we define the stronger notion of a complete precedence system, which is an unambiguous system that utilizes all possible “superfluous” parentheses. We also give a characterization of such systems in Theorem 5.2.

4. Finally, in Section 6, we closely analyze the structures of complete systems and show in Theorem 6.5 that complete systems satisfy the assumption of having all the operators partitioned in a chain of left associative and right associative classes, and get some extra restrictions on the relative positions and the associativity of unary operators.
2 Basic Definitions

2.1 Formal Languages

Define \( N = \{0, 1, 2, \ldots \} \), where, following Von Neumann’s notation, each \( n \in N \) is defined by \( n = \{0, 1, \ldots, n-1\} \). Thus, \( 0 = \{\} = \emptyset, 1 = \{\emptyset\}, \) etc.

An alphabet \( \Sigma \) is a nonempty set. Elements of \( \Sigma \) are called letters. For two sets \( A \) and \( B \), \( B^A \) denotes the set of all functions from \( A \) to \( B \), each function being viewed as a subset of the Cartesian product \( A \times B \).

A word \( w \) over \( \Sigma \) of length \( n \) is an element of \( \Sigma^n \), i.e. \( w : n \to \Sigma \). Since \( n = \{0, 1, \ldots, n-1\} \) has a natural order, \( w \) will be denoted by \([w[0], w[1], \ldots, w[n-1]]\), or simply by \( w[0]w[1]\ldots w[n-1]\). Let \( e \) denote the empty word, i.e. the (unique) word over \( \Sigma \) of length 0. In fact, \( e = \emptyset \), since \( \emptyset \) is the only function from \( \emptyset \) to \( \Sigma \).

Let \( \Sigma^* = \bigcup_{n \in N} \Sigma^n \) denote the set of all words over \( \Sigma \). A subset \( L \subseteq \Sigma^* \) is called a language over \( \Sigma \).

For two words \( u \in \Sigma^n, v \in \Sigma^m \), the concatenation \( uv \in \Sigma^{n+m} \) is defined by:

\[
[uv][i] = \begin{cases} u[i] & \text{if } 0 \leq i < n \\ v[i-n] & \text{if } n \leq i < n + m \end{cases}
\]

Hence, \( uv \) is just the juxtaposition of \( u \) and \( v \) in one word.

For a set \( A \), \( |A| \) denotes its cardinality. Thus, for a word \( w \in \Sigma^n \), \( |w| = n \) denotes its length. Obviously, \( |uv| = |u| + |v| \).

For a letter \( a \in \Sigma \) and a word \( w \) over \( \Sigma \) we let \( n_a[w] \) denote the number of occurrences of \( a \) in \( w \). It then follows that \( n_a[uv] = n_a[u] + n_a[v] \).

If \( w = uv \), then \( u \) is called a prefix of \( w \), and \( v \) is called a suffix of \( w \). Moreover, if \( v \neq e \), \( u \) is called a proper prefix of \( w \), and if \( u \neq e \), \( v \) is called a proper suffix of \( w \). If \( w = xuy \), \( u \) is called a subword of \( w \).

The following proposition follows from the definitions.

**Proposition 2.1**  For all words \( u, v, x, y \in \Sigma^* \), the following is true:

1) The right and left cancelation laws hold, i.e.
   
   if \( ux = vx \) or \( xu = vx \), then \( u = v \).
2) If \( uv = xy \), then
   
   • \( u \) is a prefix of \( x \) or \( x \) is a prefix of \( u \) and
   
   • \( v \) is a suffix of \( y \) or \( y \) is a suffix of \( v \).

\[ \blacksquare \]

2.2 Grammars

Recall that a language is just a set of words. A language over the alphabet \( \Sigma \) is sometimes defined by a context-free grammar, which is a quadruple \( [\Sigma, \mathcal{V}, S, \mathcal{P}] \), where \( \mathcal{V} \) is a set of variables that is disjoint from \( \Sigma \); \( S \in \mathcal{V} \) is called the starting symbol, and \( \mathcal{P} \) is a set of production rules of the form

\[ V \rightarrow u, \]

where \( V \in \mathcal{V} \) and \( u \in [\Sigma \cup \mathcal{V}]^* \). The idea behind grammars is that, intuitively, in order for a word \( w \in \Sigma^* \) to belong to the language defined by the grammar, we should be able to derive \( w \) from the starting symbol \( S \) using the production rules in \( \mathcal{P} \). In other words, we can construct

\[ \text{\textsuperscript{1}}\text{Note that in our metalanguage we sometimes use the brackets } [\cdot ] \text{ instead of the parentheses } (\cdot ), \text{ when the latter may appear in the object language.} \]

\[ 3 \]
a sequence of words \( w_0, \ldots, w_n \) (called a derivation), where \( w_0 = S, w_n = w \), and each \( w_{i+1} \) results from \( w_i \) when applying a production rule \( V \rightarrow u \) in \( P \), i.e. replacing some occurrence of the variable \( V \) in \( w_i \) by the word \( u \).

A derivation of a word \( w \) using a grammar can also be seen as a labeled tree (called the derivation tree or parsing tree) defined as follows:

1. The root is labeled \( S \);
2. Suppose that a node \( v \) is labeled by a variable \( V \), and the derivation uses a production rule of the form \( V \rightarrow u \) to replace \( V \) by \( u = u_0 \ldots u_{n-1} \). If \( n > 0 \), \( v \) has exactly \( n \) children labeled \( u_0, \ldots, u_{n-1} \) (from left to right), and if \( n = 0 \) (i.e. \( u = e \)), \( v \) has exactly one child labeled \( e \).

Thus, the word \( w \) can be read off the labels of all the leaves of the parsing tree.

The idea behind a parsing tree is to unify all derivations that are essentially the same (up to the order of the production rules used). Thus, a parsing tree of a word \( w \) gives the syntactical structure of \( w \). Since we like to have a unique syntactical structure for each word, we sometimes require that a grammar defining a language does not have two different parsing trees for the same word. Such a grammar is called unambiguous. A language which can be generated by some unambiguous grammar is called unambiguous.

2.3 The Language of Expressions

We now consider general expressions in infix notation that are built using only unary and binary operators starting from a nonempty set of atomic expressions. Thus the alphabet \( \Sigma \), over which general expressions are defined, is defined as follows:

\[
\Sigma = A \cup C \cup \{(,\})
\]

where \( A \) is a nonempty set of atomic expressions, \( \{(,\}) \) is the set of parentheses used, and the set of operators \( C \) is the disjoint union of the sets \( U_L \), \( U_R \) and \( B \) of the left unary, right unary, and binary operators respectively.

The set \( E \subseteq \Sigma^* \) of well formed expressions (or simply expressions) is defined by the context free grammar below, where \( S \) is the starting symbol and the only variable, and the production rules are:

\[
S \rightarrow a \text{, for each } a \in A;
S \rightarrow (\circ S) \text{, for each } \circ \in U_L;
S \rightarrow (S\circ) \text{, for each } \circ \in U_R;
\]

and

\[
S \rightarrow (S \ast S) \text{, for each } \ast \in B.
\]

We note here the use of parentheses to resolve the syntax of the expressions.

In proving statements about expressions \( E \in E \) we will sometimes use induction on the number of applications of production rules. We then say that we use induction on expressions.

Examples:

1. Formulas of Propositional Logic: Here \( A \) is the set of atomic formulas together with True and False, \( U_L = \{\} \), \( U_R = \emptyset \) and \( B = \{\land, \lor, \rightarrow, \leftrightarrow\} \).

2. Arithmetic Expressions: Here \( A \) is the set of variables and constants, \( U_L = \{\oplus\} \), \( U_R = \emptyset \) and \( B = \{+, -, \times, \div, \uparrow\} \), where \( \oplus \) stands for unary negation and \( \uparrow \) stands for exponentiation.\(^2\)

\(^2\)Note that the unary negation “\( \oplus \)” is different form the binary subtraction operator “\( - \)”.
3. Regular Expressions: Here $A$ is the set of all $\pi$ (denoting $\{a\}$, where $a$ is a letter in the underlying alphabet) together with the symbols $\emptyset$ (denoting $\emptyset$) and $\pi$ (denoting $\{e\}$), $U_\emptyset = \emptyset$, $U_\pi = \{\ast\}$ (the Kleene-star operation), and $B = \{+, \cdot\}$ (denoting the union and the concatenation operations).

The way parentheses are introduced in expressions motivates the following definition:

**Definition 2.1** [2] A word $\alpha \in \Sigma^*$ is parenthetically balanced or simply balanced if the following two conditions hold:
1) $n_1(\alpha) = n_1(\alpha)$
2) For each prefix $\beta$ of $\alpha$, $n_1(\beta) \geq n_1(\beta)$.

We say that $\alpha$ is properly balanced if it satisfies (1) and the stronger condition:
3) For each nonempty proper prefix $\beta$ of $\alpha$, $n_1(\beta) > n_1(\beta)$.

The following lemma easily follows from Definition 2.1:

**Lemma 2.2** Let $\alpha, \beta, \gamma \in \Sigma^*$. The following holds:
(i) If $\alpha$ is balanced, then for each suffix $\beta$ of $\alpha$, $n_1(\beta) \leq n_1(\beta)$.
(ii) If $\alpha = \beta\gamma$ and if two of these words are balanced, then so is the third.

Note the special case when one of $\beta$ or $\gamma$ does not contain any parentheses, in which case it is trivially balanced. In contrast of (ii) we have:
(iii) If $\alpha$ is properly balanced, then none of its nonempty proper prefixes or suffixes is.
(iv) If $\alpha\beta\gamma$ is balanced (properly balanced), then so is $\alpha(\beta)\gamma$.
(v) If $\beta$ and $\alpha(\beta)\gamma$ are balanced, then so is $\alpha\beta\gamma$.

(Note the special case when $\alpha$ or $\gamma = e$.)
(vi) If $\beta$ is balanced, $\alpha, \gamma \neq e$, and $\alpha(\beta)\gamma$ is properly balanced, then so is $\alpha\beta\gamma$.
(vii) $\beta$ is balanced iff $\beta$ is properly balanced.

**Corollary 2.3** Let $E \in \mathcal{E}$ be an expression. Then:
1. $E$ is properly balanced.
2. Every properly balanced subword $E'$ of $E$ is also an expression.
3. If $E'$ results from $E$ when some pairs of matching parentheses are removed, then $E'$ is balanced. Such $E'$ is called a quasi-expression.

**Proof:** By induction on expressions and applying Lemma 2.2.

**Corollary 2.4** The grammar above defining the set of expressions $\mathcal{E}$ is unambiguous.

**Proof:** By induction again and applying Corollary 2.3. The idea is that given an expression $E$, one can uniquely determine the first production rule used in the derivation of $E$, together with the subexpression(s) mentioned in the right hand side of the rule. The fact that an expression is properly balanced will be used to determine where it ends, if we know where it begins (or vice versa).

The only operator occurring on Level 1 of the unique parsing tree of an expression $E$ will be called the main operator of $E^3$.

---

3Note that the root at Level 0 is always labeled by the starting symbol $S$. The reader may be used to a slightly different convention where the root gets labeled by the main operator. However, this convention is useful only when we abstract the expression from its linear syntactical form.
3 Superfluous Parentheses

Parentheses are sometimes omitted to improve readability. For example the quasi-expression $a + b \times c$ is sometimes written instead of the full expression $(a + (b \times c))$, where we rely on some intuition needed to recover parentheses. Likewise, $a \times b + c$ is written instead of $((a \times b) + c)$. Here we use the convention that $\times$ precedes $+$ (when we try to evaluate the expression)\(^4\).

In the literature of programming languages, $\times$ is referred to have a “higher precedence” or “higher binding strength” than $\div$. In this paper, however, we will not use such a terminology, as we visualize the precedence order as a precedence relation between numbers (e.g. $1$ precedes $2$). In fact, we will see in Section 6, that we can sometimes map the operators $\times$ and $\div$ to natural numbers in a way that respects the precedence relation.

Note that the outside parentheses should generally be kept if $(a + b \times c)$ is a subexpression of a larger one. However, the outermost parentheses of any expression are also superfluous, and can safely be removed.

Next, we note that an expression like $((a \div b) \div c)$ may sometimes be abbreviated by $a \div b \div c$, while $(a \uparrow (b \uparrow c))$ is abbreviated by $a \uparrow b \uparrow c$. Here we say that $\div$ is left associative while $\uparrow$ is right associative.\(^5\)

The concept of left and right associativity can also be used to resolve conflicts between some operators. E.g. the quasi-expression $a - b + c$ abbreviates $(a - b) + c$, while $a + b - c$ abbreviates $(a + b) - c$. Here, none of the operators $+$ and $-$ is declared to precede the other, but together they associate to the left.

One can now ask questions like:

1. Can we have three operators $*1, *2, *3$, such that $*1$ precedes $*2$, $*2$ precedes $*3$ and $*3$ precedes $*1$?
2. Can we have two operators $*1, *2$ that associate to the left, but $*1$ is right associative with itself or with a third operator $*3$?
3. How should a left unary operator compare to binary operators, to other left unary operators, or to right unary operators?

To make our discussion general enough, we start with a definition of a so-called precedence system, which declares for each expression of the form $((E_1 \circ E_2) \circ E_3)$ or $(E_1 \circ (E_2 \circ E_3))$ (where the parenthesis pairs shown are matching), if the inner parenthesis pair is superfluous. The system’s decision should be based only on the operators $\circ, \ast$ as well as their relative positions, i.e. the decision should be independent of the subexpressions $E_1, E_2$ and $E_3$. Thus, we are lead to the following:

**Definition 3.1** Consider an expression alphabet $\Sigma = A \cup C \cup \{(,),\}$, where $A$ is the set of atomic expressions, and $C$ is the disjoint union of the sets $U_L$, $U_R$ and $B$ of the left unary, right unary, and binary operators respectively.

A precedence system on $C$ is a pair of relations $S = [\prec_L, \prec_R]$, where

\[
\prec_L \subseteq C \times [B \cup U_R] \text{ and } \prec_R \subseteq C \times [B \cup U_L],
\]

i.e. $\prec_L$ and $\prec_R$ are binary relations on $C$ with some restrictions on their images (as will be clarified below).

\(^4\)The reader may prefer to say that $\times$ precedes $\div$, but this, of course, depends on the way we view an expression, whether it is a tree growing downward or upward, and whether we evaluate that tree by first evaluating its leaves and then proceed to the root (or vice versa).

\(^5\)I always explain to my students the wisdom behind the right associativity in an expression like $e^{x^2}$, namely, we don’t really mean $e^{2x}$. This, of course, does not rule out that some CS scholars prefer all operators to be left associative.
Let \( S = [\prec_L, \prec_R] \) be a precedence system on \( C \), and \( \circ, * \) be two operators in \( C \). Also, let \( E \) be an expression over \( \Sigma \), with the subexpression:

\[
((E_1 \circ E_2) \circ E_3)
\]

where the two pairs of parentheses shown are matching, and \( E_1, E_2, E_3 \) are expressions. Then the inner pair of matching parentheses is called \( S \)-superfluous (or simply superfluous if \( S \) is understood) iff \( \circ \prec_L * \).

To cover the cases when either (or both) of the operators \( \circ, * \) is unary, we also allow some of the words \( E_1, E_2, \) and \( E_3 \) to be empty. However, we assume here that \( * \) takes a left argument, and therefore could not be left unary. (This is the reason behind the restriction on the image of the binary relation \( \prec_L \).)

Similarly, the inner pair of matching parentheses in \((E_1 \circ (E_2 \circ E_3))\) is \( S \)-superfluous iff \( \circ \prec_R * \). Here also, \( * \) is not allowed to be right unary.

Finally, the outermost parentheses in \( E \) are also superfluous.

Intuitively, \( \circ \prec_L * \) states that \( \circ \) precedes \( * \) from the left side, while \( \circ \prec_R * \) states that \( \circ \) precedes \( * \) from the right side.

When some pairs of superfluous parentheses are removed from an expression, the resultant word is called an \( S \)-quasi-expression of \( E \). When all superfluous parentheses are removed, the resultant word is called the \( S \)-succinct quasi-expression of \( E \).

Note that the definition of superfluous parentheses does not depend on the context. More precisely:

**Proposition 3.1** Let \( S \) be a precedence system. The language of all \( S \)-succinct expressions is context free.

**Proof:** We give a context-free grammar \( G \) generating the language of succinct expressions. The grammar \( G \) has the starting symbol \( S \) and the variables \( S_\circ, \) for each operator \( \Delta \in C \). Informally, \( S_\circ \) will generate a succinct expression with \( \Delta \) as its main operator. The production rules are thus:

\[
\begin{align*}
S & \rightarrow a \quad \text{for each } a \in A; \\
S & \rightarrow S_\circ \quad \text{for each } \Delta \in C \\
S_\circ & \rightarrow \Delta a \quad \text{for each } \Delta \in U_\Delta; \ a \in A \\
S_\circ & \rightarrow \Delta S_\star \quad \text{for each } \Delta \in U_\star; \ * \in C, \text{ such that } * \prec_R \Delta \\
S_\circ & \rightarrow \Delta (S_\star) \quad \text{for each } \Delta \in U_\star; \ * \in C, \text{ such that } * \neq_R \Delta \\
S_\circ & \rightarrow a \Delta \quad \text{for each } \Delta \in U_\Delta; \ a \in A \\
S_\circ & \rightarrow S_\circ \Delta \quad \text{for each } \Delta \in U_\star; \ * \in C, \text{ such that } * \prec_L \Delta \\
S_\circ & \rightarrow (S_\circ) \Delta \quad \text{for each } \Delta \in U_\star; \ * \in C, \text{ such that } * \neq_L \Delta \\
S_\circ & \rightarrow a \Delta a' \quad \text{for each } \Delta \in B; \ a, a' \in A \\
S_\circ & \rightarrow a \Delta S_\star \quad \text{for each } \Delta \in B; \ a \in A; \ * \in C, \text{ such that } * \prec_R \Delta \\
S_\circ & \rightarrow a \Delta (S_\star) \quad \text{for each } \Delta \in B; \ a \in A; \ * \in C, \text{ such that } * \neq_R \Delta \\
S_\circ & \rightarrow S_\circ \Delta a \quad \text{for each } \Delta \in B; \ a \in A; \ * \in C, \text{ such that } * \prec_L \Delta \\
S_\circ & \rightarrow S_\circ \Delta S_\star \quad \text{for each } \Delta \in B; \ * \in C, \text{ such that } * \neq_L \Delta, * \neq_R \Delta \\
S_\circ & \rightarrow (S_\circ) \Delta S_\star \quad \text{for each } \Delta \in B; \ * \in C, \text{ such that } * \neq_L \Delta, * \neq_R \Delta \\
S_\circ & \rightarrow (S_\circ) \Delta (S_\star) \quad \text{for each } \Delta \in B; \ * \in C, \text{ such that } * \neq_L \Delta, * \neq_R \Delta
\end{align*}
\]
Note that each derivation of a succinct quasi-expression in the above grammar $G$ keeps track of the structure of the original expression, i.e. to recover the original expression, we just insert all missing parentheses in the derivation. This motivates the following:

**Definition 3.2** If a quasi-expression $E'$ is the $S$-succinct quasi-expression of two distinct expressions, $E'$ is called $S$-ambiguous. Note that an ambiguous quasi-expression exists iff the grammar $G$ of Proposition 3.1 is ambiguous. In this case, we say that the precedence system $S$ is ambiguous.

We note here that unambiguous precedence systems are easy to find. E.g. consider the system $[\emptyset, \emptyset]$, which does not allow any omission of parentheses (except the outermost ones). Such a system is definitely unambiguous, but not very useful.

How do unambiguous systems in general look like? It turns out that they could behave in a very strange way. Theorem 3.4 below gives a characterization of unambiguous precedence systems.

However, we first need to understand the quasi-expressions in $S$. To this end, we start with the following:

**Definition 3.3** Let $\alpha$ be a parenthesis-free word over $A \cup C$, we define the following two sets of operators $C_L[\alpha]$ and $C_R[\alpha]$ as follows:

If $\alpha$ is the empty word $e$, set $C_L[e] = U_R$ and $C_R[e] = U_L$ (this is useful in the recursive step below). If $\alpha$ is an atomic expression $a$, set $C_L[a] = B \cup U_L$ and $C_R[a] = B \cup U_R$. Otherwise, recursively define:

$$C_L[\alpha] = \{ \epsilon \in [B \cup U_L] : \alpha = \beta \star \gamma, \text{ for some } \beta, \gamma, \text{ where } \star <_R \circ \text{ and } \star \in C_R[\beta] \cap C_L[\gamma] \}$$

$$C_R[\alpha] = \{ \epsilon \in [B \cup U_R] : \alpha = \beta \circ \gamma, \text{ for some } \beta, \gamma, \text{ where } \circ <_L \star \text{ and } \circ \in C_R[\beta] \cap C_L[\gamma] \}$$

Informally, if $\alpha$ is nonempty, the set $C_L[\alpha]$ ($C_R[\alpha]$) contains the possible operators that can be safely attached to the left (right) of $\alpha$ (i.e. yielding a quasi-expression with $\alpha$ as a sub-quasi-expression). This is made more rigorous by the following lemma, which has a straightforward inductive proof:

**Lemma 3.2** Let $\alpha$ be a nonempty parenthesis-free word over $A \cup C$. If either one of the sets $C_L[\alpha]$ or $C_R[\alpha]$ is nonempty, then $\alpha$ is the $S$-succinct quasi-expression of some expression. Moreover, $C_L[\alpha]$ (or $C_R[\alpha]$) contains exactly those operators that allow the removal of the outer parentheses around $\alpha$ when connecting it from the left (or right). More precisely:

1. $\star \in C_R[\alpha]$ iff the parentheses in the quasi-expression $(\alpha) \star E$ are superfluous, and

2. $\star \in C_L[\alpha]$ iff the parentheses in the quasi-expression $E \star (\alpha)$ are superfluous.

where $E$ stands for any expression (or the empty word if $\star$ is unary).

We also get the following:

**Lemma 3.3** Let $\alpha$ be a parenthesis-free $S$-succinct quasi-expression.

1. If we can write $\alpha = \beta_1 \star_1 \gamma_1 = \beta_2 \star_2 \gamma_2$ in two different ways (i.e. $\beta_1 \neq \beta_2$), such that for each $i = 1, 2, \star_i \in C_R[\beta_i] \cap C_L[\gamma_i]$, then $\alpha$ is ambiguous.

2. If $\alpha$ is ambiguous, then it contains a subword $\alpha'$ that can be written as $\alpha' = \beta_1 \star_1 \gamma_1 = \beta_2 \star_2 \gamma_2$ in two different ways (i.e. $\beta_1 \neq \beta_2$), such that for each $i = 1, 2, \star_i \in C_R[\beta_i] \cap C_L[\gamma_i]$.

We can now state the following:
Theorem 3.4 Let $C = B \cup U_L \cup U_R$ be the disjoint union of binary, left unary, and right unary operators, respectively, and let $S = [\lessdot, \prec_R]$ be a precedence system over $C$. Then $S$ is ambiguous iff for some $n, m \geq 1$, there are operators $o_1, o_2, \ldots, o_n \in [B \cup U_L]$ and $*_1, *_2, \ldots, *_m \in [B \cup U_R]$ such that:

$$*_m \prec_R o_1 \prec_R o_2 \prec_R \ldots \prec_R o_n \lessdot \prec_L *_1 \prec_L *_2 \prec_L \ldots \prec_L *_m$$

(1)

Before proving the theorem, let us visualize a precedence system $S = [\lessdot, \prec_R]$ over $C$ as describing a directed graph over $C$ with edges of two colors, red for $\prec_R$, and lemon yellow for $\lessdot$. Thus, the theorem states that $S$ is ambiguous iff there is a cycle in the graph that can be decomposed into a red path followed by a yellow path.

Proof: The if direction is the easy part. If the system $S$ has the prescribed precedence chain, then the quasi-expression $o_n \ldots o_2 o_1 a *_1 *_2 \ldots *_m$ is ambiguous, as it abbreviates both $(((o_n \ldots (o_2 o_1 a)) *_1 *_2) \ldots *_m)$ and $(o_n \ldots (o_2 o_1 ((a *_1) *_2) \ldots *_m)))$, where for clarity we suppressed all left arguments (if any) of the operators $o_i$, and all right arguments (if any) of the operators $*_{i}$.

For the only-if direction, we assume that the system $S$ is ambiguous and show that there is a cycle of the form (1).

Let $\alpha$ be an ambiguous quasi-expression of minimal length. Then $\alpha$ must be parenthesis-free. If not, we could replace in $\alpha$ any subword of the form $(\beta)$, where $\beta$ is parenthesis-free, by an atomic expression, to get a shorter ambiguous quasi-expression.

Choose in $\alpha$ a subword $\beta \circ \gamma \circ \delta$ with minimal $|\gamma|$, such that $\circ \in C_L[\gamma \circ \delta]$ and $\circ \in C_R[\beta \circ \gamma]$. Such a subword exists from Part 2 of Lemma 3.3. We now claim that there are two chains connecting $\circ$ and $*$ of the form:

$$* = *_1 \lessdot L *_2 \lessdot L \ldots \lessdot L *_m \prec_R 0,$$

and

$$\circ = o_1 \prec_R o_2 \prec_R \ldots \prec_R o_n \lessdot \prec_L \circ.$$

Using symmetry, we only show the existence of the first chain. Using the fact that $\circ \in C_L[\gamma \circ \delta]$, we know that $\gamma \circ \delta$ can be written as $\gamma' \circ' \delta'$, where $\circ' \prec_R \circ$ and $\circ' \in C_R[\gamma] \cap C_L[\delta']$.

If $\gamma = \gamma'$, then $* = \circ'$ $\prec_R \circ$, and we are done.

If $|\gamma'| < |\gamma|$ (i.e. $\gamma'$ is a proper prefix of $\gamma$), we can then write $\beta \circ \gamma \circ \delta = \beta \circ \gamma' \circ' \gamma'' \circ \delta$, with $\circ' \in C_L[\gamma' \circ \delta]$, contradicting the minimality of $|\gamma|$.

We are now left with the case when $|\gamma| < |\gamma'|$, i.e. $*'$ occurs to the right of $*$. Here we consider the longest $\lessdot_L$-chain of operators: $*_1 \lessdot L *_2 \lessdot L \ldots \lessdot L *_m = *'$, such that $\gamma \circ \delta = \gamma' \circ' \delta' = \gamma_1 *_1 \gamma_2 *_2 \ldots *_m \circ' *_m \circ' \delta'$.

If $|\gamma_1 *_1 \gamma_2| > |\gamma|$ (i.e. $* \circ$ occurs to the left of $*_{2}$), and $*_{1} \in C_R[\gamma_1] \cap C_L[\gamma_2]$. From the fact that $\circ' \in C_R[\gamma']$ and the definition of $C_R$, we know that $m \geq 2$.

We now show that $*_{1} = *$.

If $*_{1}$ occurs to the right of $*$, we can use the fact that $*_{1} \in C_R[\gamma_1]$ to make one more step to the left and write $\gamma_1 = \gamma_0 \circ*_{1} \gamma'_{1}$, with $\circ*_{1} \circ \in C_R[\gamma_0] \cap C_L[\gamma_{1}']$, contradicting the maximality of $m$.

If $*_{1}$ occurs to the left of $*$, which occurs to the left of $*_{2}$, we can decompose $\gamma_2 = \gamma' * \gamma''$, and then get the subword $\beta \circ \gamma_1 *_{1} \gamma' * \gamma''$, with $*_{1} \in C_L[\gamma' * \gamma'']$ and $* \in C_R[\beta \circ \gamma_1 *_{1} \gamma']$, again contradicting the minimality of $|\gamma|$.

Thus, we must have $*_{1} = *$, concluding the proof of the theorem.

Before proceeding to the applications of Theorem 3.4, let us get some insight of its proof by stating an abstract version of the theorem. We first need few definitions:
**Definition 3.4** Let $V$ be a nonempty set of vertices. We define an ordered graph $(V, E, <)$ to be a directed graph, i.e. $E \subseteq V \times V$, equipped with a linear order $<$ on $V$.

We also recursively define: An ordered graph $(V, E, <)$ is an ordered binary tree rooted at $r \in V$ iff either $V = \{r\}$, or the ordered subgraphs induced by the left and right subintervals $\{v \in V : v < r\}$ and $\{v \in V : v > r\}$ are themselves ordered binary trees rooted at $r_1 < r$ and $r_2 > r$, respectively, and the two edges $(r, r_1), (r, r_2)$ belong to $E$.

Thus, an ordered binary tree is a tree structure that uses (some of) the edges in $E$ to connect a parent to its children, and respects the linear order in the sense that, under each internal node the left subtree is $<$ the right subtree. In other words, each node in the tree is either a leaf (single), or a parent $p$ connected to exactly two children $b$ (boy) and $g$ (girl), such that $b < p < g$, and

“each descendent from $b$" $<$ “each descendent from $g$".

To see the connection between the above definitions and the language of quasi-expressions, we state the following proposition, which has an easy inductive proof.

**Proposition 3.5** Let $E = e_1e_2...e_l$ be a parenthesis-free quasi-expression of length $l$, and let $(V, E, <)$ be the ordered graph induced by $E$ as follows: $V = \{1, 2, \ldots, l\}$, $<$ is the natural order, and $(n, m) \in E$ iff

- $(n < m$ and $(e_m <_R e_n$ or $(e_m \in A$ and $e_n \in (B \cup U_L))$) or
- $(n > m$ and $(e_m <_L e_n$ or $(e_m \in A$ and $e_n \in (B \cup U_R))$)

Then the above ordered graph induced by $E$ is an ordered binary tree rooted at $n$ iff $E$ is the succinct quasi-expression of an expression with main operator $e_n$.

Note that the edge relation $E$ extends the union of the inverse of the precedence relations $<_L$ and $<_R$ by letting all operators point to the atomic expressions in their possible scopes.

**Example:** The quasi-expression $a + b \times c$ viewed as an ordered graph can be made into an ordered binary tree rooted at (the position of) “+” with its left subtree $a$ and right subtree $b \times c$, which are themselves ordered binary trees. E.g. $a$ is rooted at (the position of) “a”, while $b \times c$ is rooted at (the position of) “×”. Note the edges connecting the positions of “+” to “×” to “b”.

We are now ready to state the following theorem which has essentially the same proof as that of Theorem 3.4.

**Theorem 3.6** An ordered graph, that can be made into two ordered binary trees with different roots, must have two elements $v_1, v_2$, such that there is an increasing path from $v_1$ to $v_2$, and a decreasing path from $v_2$ to $v_1$.

From the discussion in the beginning of Section 3, recall that two operators $\ast, \circ$ are called left associative with each other iff the parentheses in both of the expressions $(a \ast a) \circ a$ and $(a \circ a) \ast a$ are superfluous, i.e. both $\ast <_L \circ$ and $\circ <_L \ast$. Right associativity among operators is similarly defined. From Theorem 3.4 we learn the following:

**Corollary 3.7** An unambiguous precedence system does NOT allow any of the following:

1. An operator $\ast$ is left associative with itself but right associative with another operator.

2. An operator $\ast$ is left associative with an operator $\circ$ but right associative with another operator $\triangle$.

We note here that a succinct quasi-expression of an ambiguous precedence system might not be ambiguous. Thus, it is interesting to know when such a quasi-expression is ambiguous. Also, given an unambiguous quasi-expression, it is important to get its main operator (for parsing purposes). These tasks are achieved by the following:
Let \( |C| = m \), and \( S \) be a precedence system over \( C \). There is an algorithm that takes any quasi-expression \( E \) of length \( n \), and decides in time \( O(mn^3) \) and space \( O(mn^2) \) whether \( E \) is ambiguous, and moreover produces the positions of all possible main operators of \( E \).

**Proof:** We first use the notation \( E[i,j] \) to denote the subword of \( E \) of length \( j \) starting at the \( i \)-th position. Thus, \( E[1,n] = E, E[i,1] \) is the letter in the \( i \)-th position, and \( E[i,0] = \epsilon \) (the empty word). Our algorithm (given in the appendix) is a dynamic program, which calculates the sets \( C_L[i,j] = C_L[E[i,j]] \) and \( C_R[i,j] = C_R[E[i,j]] \) of Definition 3.3 together with the set \( M[i,j] \) of the positions of possible main operators for \( E[i,j] \). These sets are calculated starting with \( j = 0, 1, \ldots, n - 1 \), where in every stage the set of smaller values of \( j \) are used to decide whether the subword \( E[i,j] \) is a quasi-expression, whether it is ambiguous, and whether \( k = i, \ldots, i + j - 1 \) is a position of a possible main operator for \( E[i,j] \). The algorithm must run through all \( i, j, k \) and all operators in \( C \). Thus it takes time \( O(mn^3) \). Also, it only needs a storage for the counters and the sets \( C_L[i,j], C_R[i,j], M[i,j] \), which can take as much as \( O(mn^2) \) space.

4 Maximal Unambiguous Systems

Let us ask the following question: How far can we extend \( \prec_L, \prec_R \) to allow the omission of as many pairs of matching parentheses as possible without stepping into the ambiguity?

**Definition 4.1** Let \( S, S' \) be two precedence systems on \( C \), we say that \( S \) is a restriction of \( S' \) (or \( S' \) is an extension of \( S \)), written \( S \leq S' \), iff in each expression, each pair of \( S \)-superfluous parentheses is also \( S' \)-superfluous.

When do we have \( S \leq S' \)? The answer is given by:

**Proposition 4.1** Let \( S = [\prec_L, \prec_R], S' = [\prec'_L, \prec'_R] \) and \( S'' = [\prec''_L, \prec''_R] \). Then
1. \( S \leq S' \) iff both \( \prec_L \leq \prec'_L \) and \( \prec_R \leq \prec'_R \).
2. \( S \leq S' \leq S'' \) iff \( S = S' \) (i.e. \( \prec_L = \prec'_L \) and \( \prec_R = \prec'_R \)).
3. If \( S \leq S' \leq S'' \), then \( S \leq S'' \).

Note that (2) and (3) state that the relation \( \leq \) is a partial order on the set of all precedence systems.

**Proof:** (2) and (3) follow easily from (1). For (1), the if direction is obvious. The only-if direction follows from the fact that if \( \prec_L \preceq \prec'_L \) (say), then there are operators \( o, * \) such that \( o \prec_L * \) but \( o \not\prec'_L * \), in which case the parentheses in \( (a \circ a) \ast a \) are \( S \)-superfluous but not \( S' \)-superfluous.

We note here that restrictions of unambiguous precedence systems are themselves unambiguous:

**Proposition 4.2** If \( S \leq S' \) and \( S' \) is unambiguous, then so is \( S \).

This motivates the following:

**Definition 4.2** A precedence system \( S \) is called maximal unambiguous iff it is unambiguous and \( \leq \)-maximal among the unambiguous systems, i.e. whenever \( S \leq S' \) and \( S' \) is unambiguous, then \( S = S' \).

How can we quickly tell if a precedence system is maximal unambiguous? This question is answered by the next theorem:

**Theorem 4.3** A precedence system \( S = [\prec_L, \prec_R] \) is maximal unambiguous iff the following conditions hold:
1. Both of the relations \( \prec_L \) and \( \prec_R \) are transitive.

2. For each pair of operators \( \circ, \ast \in \mathcal{C} \), \( \ast \prec_L \circ \) iff none of the following is true:
   
   (a) \( \ast \prec_L \circ \).
   
   (b) There exists \( \square \in \mathcal{C} \), such that \( \ast \prec_L \circ \square \) or \( \ast \prec_L \square \circ \).
   
   (c) There exists \( \triangle \in \mathcal{C} \), such that \( \ast \prec_L \triangle \circ \).

3. Exactly like (2) with \( \prec_L \) and \( \prec_R \) switched.

**Proof of the only-if part:** Let \( S \) be maximal unambiguous.

1. If \( \prec_L \) is not transitive, then there are some operators \( \circ, \square \in \mathcal{C} \) such that \( \circ \prec_L \ast \prec_L \square \), but \( \circ \not\prec_L \square \). Extend \( S \) to \( S' = [\prec_L \cup \{\circ, \square\}, \prec_R] \). Since \( S \) is maximal unambiguous, \( S' \) must be ambiguous. Using Theorem 3.4, there must be a yellow-red cycle in \( S' \), which must use the edge \( \circ \prec_L \square \). We can now replace that edge with the two edges \( \circ \prec_L \ast \prec_L \square \), and get a yellow-red cycle in \( S \), contradicting the unambiguity of \( S \). Thus, \( \prec_L \) is transitive. A similar argument holds for \( \prec_R \).

2. For the if part, note that if \( \circ \prec_L \ast \), and any one of (a), (b), or (c) holds, then we get the forbidden yellow-red cycle, contradicting the unambiguity of \( S \).

   For the if part, assume \( \circ \not\prec_L \ast \). Extend \( S \) to \( S' = [\prec_L \cup \{\circ, \ast\}, \prec_R] \). As before, \( S' \) must be ambiguous, and we can use Theorem 3.4 to get a yellow-red cycle in \( S' \) that uses the edge \( \circ \prec_L \ast \). If we remove this edge, the remaining path from \( \ast \) to \( \circ \) must be either red, yellow-red, red-yellow, or yellow-red-yellow. Now using the transitivity of \( \prec_L \) and \( \prec_R \) of (1), each yellow or red path can be shrunk into a single edge, giving (a), (b), or (c).

3. Similar to (2).

**Proof of the if part:** Assume Conditions 1-3. We first show that \( S \) is unambiguous. If not, use Theorem 3.4 to get a yellow-red cycle, and use (1) to shrink that cycle to one of the form \( \circ \prec_L \ast \prec_R \circ \). Next, we show that \( S \) is maximal. Let \( S' \) be an unambiguous extension of \( S \) by exactly one edge. If the extra edge has the form \( \circ \prec_L \ast \), then \( \circ \not\prec_L \ast \). From (2), one of (a), (b) or (c) holds for \( S \), and consequently for \( S' \), which then contradicts the unambiguity of \( S' \). Similarly for the edge being \( \circ \prec_L \ast \).

**Corollary 4.4** If \( S \) is maximal unambiguous, then for each operator \( \ast \in \mathcal{C} \), exactly one of \( \ast \prec_L \circ \) or \( \ast \prec_R \circ \) holds.

In particular, if \( \ast \) is right unary, then \( \ast \prec_L \circ \), and if \( \ast \) is left unary, then \( \ast \prec_R \circ \).

**Proof:** Since \( S \) is unambiguous, we can not have both \( \ast \prec_L \ast \prec_R \ast \) (a yellow-red cycle). If \( \ast \not\prec_L \ast \), then use \( \circ = \ast \) in Condition (2) of Theorem 4.3 to get that at least one of (a), (b), or (c) holds. But with \( \circ = \ast \), each of (b) and (c) contradicts the unambiguity of \( S \). Thus (a) holds, i.e. \( \ast \prec_R \ast \).

We note here that every unambiguous system \( S \) can be extended to a maximal unambiguous one.

**Example 4.1** Consider the set of binary operators \( \mathcal{C} = \mathcal{B} = \{\circ, \ast, \square, \triangle\} \), and let \( S \) be the precedence system defined by \( \prec_L = \{\circ, \ast\}, [\square, \triangle]\) and \( \prec_R = \{\ast, \square, \triangle, \circ\} \). In other words, the graph on \( \mathcal{C} \) has just the cycle \( \circ \prec_L \ast \prec_L \circ \), \( \circ \prec_L \triangle \prec_L \circ \), which has the colors yellow, red, yellow, red. Using Theorem 3.4, we know that \( S \) is an unambiguous system, and can thus be extended to a maximal one.

Note, however, that \( S \) (and any of its maximal extensions) behave in a strange way. For example, the existence of a cycle having alternating colors makes us unable to tell which of the operators has a higher precedence.
Also, any maximal unambiguous extension of $S$ cannot allow any type of comparisons between $\circ$ and $\Box$. Thus, all parenthesis pairs in the expressions $(a \circ a) \Box a$, $a \circ (a \Box a)$, $(a \Box a) \circ a$, and $(a \Box a) \circ a$ are not superfluous, and consequently a quasi-expression like $a \Box a \circ a$ could not abbreviate any expression.

## 5 Complete Precedence Systems

Example 4.1 shows that maximal unambiguous systems may not be that attractive to deal with. It also motivates the following:

**Definition 5.1** A precedence system $S$ is called complete iff it is unambiguous and each possible quasi-expression (an expression with some matching pairs of parentheses removed) is the result of the removal of some $S$-superfluous pairs of parentheses from some expression. Equivalently, $S$ is complete iff each possible quasi-expression abbreviates a unique expression.

Thus, maximal unambiguous systems are optimal in the sense that no strict extensions of those is still unambiguous, while complete systems are optimal in the sense that they utilize all possible quasi-expressions.

The next proposition shows that the completeness property is stronger than the maximal unambiguous one.

**Proposition 5.1** Let $\mathcal{C} = \mathcal{B} \cup \mathcal{U}_L \cup \mathcal{U}_R$ be the disjoint union of binary, left unary, and right unary operators, respectively. Then:

1. Complete precedence systems are maximal unambiguous.
2. There exists a maximal unambiguous precedence system which is not complete, whenever any of the following holds:
   
   (a) $\mathcal{B} \neq \emptyset$ and $\mathcal{U}_L \cup \mathcal{U}_R \neq \emptyset$.
   
   (b) $\mathcal{U}_L \cup \mathcal{U}_R \neq \emptyset$ and $|\mathcal{U}_L \cup \mathcal{U}_R| \geq 3$.
   
   (c) $|\mathcal{B}| \geq 4$.

**Proof:**

1. Let $S$ be a complete precedence system, and let $S < S'$ (i.e. $S \leq S'$ but $S \neq S'$). Now consider a pair of matching parentheses in an expression $E$ that is $S'$-superfluous but not $S$-superfluous, and let $E'$ be the $S'$-succinct quasi-expression of $E$, i.e. $E'$ results from removing all $S'$-superfluous pairs of matching parentheses in $E$. Using the completeness of $S$, $E'$ also results from removing some $S$-superfluous pairs of matching parentheses in an expression $F$, which must be different from $E$, since some missing parenthesis pair in $E'$ is not $S$-superfluous. Now, since $S \leq S'$, $E'$ also results from removing some $S'$-superfluous pairs of matching parentheses in $F$, which shows that $S'$ is ambiguous.

2. We give an example of an incomplete maximal unambiguous system in each case:

   (a) Say we have a (left) unary operator $\circ$ and a binary operator $\ast$. Let $S$ be the system with $\prec_L = \{[\ast, \ast]\}$ and $\prec_R = \{[\ast, \circ]\}$. Then $S$ is unambiguous by Theorem 3.4, and must then have a maximal unambiguous extension $S'$. However, in $S'$ the parentheses in $a \ast (a \circ a)$ can not be superfluous, as this will complete the yellow-red cycle $\ast \prec_L \ast \prec_R \circ \prec_R \ast$. Thus, $S'$ is incomplete, as the quasi-expression $a \ast a \circ a$ does not abbreviate any expression.

   (b) Say we have (two left) unary operators $\circ, \Box$ and one (right) unary operator $\ast$. Let $S$ be the system with $\prec_L = \{[\circ, \ast]\}$ and $\prec_R = \{[\ast, \Box]\}$. Then $S$ is unambiguous and has a maximal unambiguous $S'$. Now, in $S'$ the parentheses in $\circ (\Box a)$ can not be superfluous, as this will complete the yellow-red cycle $\circ \prec_L \ast \prec_R \Box \prec_R \circ$. Thus, $S'$ is incomplete, as the quasi-expression $\circ \Box a$ does not abbreviate any expression.
Here we use the 4 operators to construct the system $S$ mentioned in Example 4.1, and get a maximal extension $S'$, which could not be complete, as mentioned at the end of the example.

We note here that a careful inspection shows that the converse of (2) is also true, i.e. if none of (a), (b), (c) holds, then maximal unambiguous systems are complete.

Similar to Theorem 4.3, we can characterize complete precedence systems by the following:

**Theorem 5.2** A precedence system $S = [\prec_L, \prec_R]$ is complete iff

1. both of the relations $\prec_L$ and $\prec_R$ are transitive; and
2. if $\circ \in [B \cup U_L] \lor * \in [B \cup U_R]$, then exactly one of $\circ \prec_L * \lor * \prec_R \circ$ holds.

In particular, if $\circ \in U_R$ and $* \in [B \cup U_R]$, then $\circ \prec_L *$.

Also, if $\circ \in [B \cup U_L] \land * \in U_L$, then $* \prec_R \circ$.

**Proof of the only-if part:** Let $S$ be complete.

1. Since, by Proposition 5.1, $S$ is maximal, we can use Theorem 4.3 to deduce (1). Alternatively, we could show (2) first, and use it to show that whenever $\circ \prec_L * \prec_L \circ$ (say), we must have $\circ \prec_L \circ$ or $\circ \prec_R \circ$ would contradict the unambiguity of $S$.

2. Assume that $\circ \in [B \cup U_L]$ (the case of $* \in [B \cup U_R]$ being similar). Consider the quasi-expression $\circ \circ \circ \circ$. If $\circ$ (or $*$) is unary, we just drop the corresponding atoms. Note that we assume that $\circ$ takes a right argument. Since this quasi-expression abbreviates exactly one of the parenthesized expressions $(\circ \circ \circ \circ)$ or $(\circ \circ \circ)$, we must have either $\circ \prec_L * \lor * \prec_R \circ$ (but not both).

Note that if $* \in U_L$, we have no choice but taking $\circ \circ \circ \circ$ to abbreviate $\circ \circ$. 

**Proof of the if part:** We assume (1) and (2). Let us show that $S$ is unambiguous. If not, use Theorem 3.4 to get a yellow-red cycle, and use (1) to shrink it to a small cycle of the form $\circ \prec_L * \prec_R \circ$, contradicting (2).

Next, we show the completeness of $S$ by showing that any possible quasi-expression $E$ (an expression with some matching pairs of parentheses removed) results, must occur at some $S$-superfluous pairs of parentheses from some expression.

If $E$ is an atom, we are done. Otherwise, we need to find the main operator of $E$.

We now assume that $E$ does not have its outermost parentheses (if $E = (E')$, we just consider $E'$ instead). Let $n_1 < n_2 < \ldots < n_k$ be the complete ordered list of the positions of all operators in $E$, that split $E$ into two (parenthetically) balanced words. Also, for $i = 1, \ldots, k$, let $\gamma_i$ be the operator occurring at the position $n_i$. Thus, for each $i$ we can write $E = \beta_i \gamma_{i-1} \ldots \gamma_1 \gamma_0$, where $\beta$, and $\gamma_i$ are balanced. (Note that the $\gamma_i$'s are not necessarily distinct.) Note that the main operator $\gamma$ of the original expression $E'$, from which $E$ results, must occur at some $n_i$, since it splits $E$ into two balanced words. Thus, the above list is not empty. However, the parenthesis pairs that were removed from $E'$ may not be superfluous, and consequently, we may need to find a (possibly different) main operator $\gamma_m$, which allows the removal of the parentheses around the subexpressions $\beta_m$ and $\gamma_m$. This is the task of the next lemma:

**Lemma 5.3** There is a unique position $n_m$, for which

1. $\gamma_m \prec_R \gamma_j$, for $1 \leq j < m$, and
2. $\gamma_m \prec_L \gamma_j$, for $m < j \leq k$.

Before proving the lemma, let us note how the lemma implies the completeness property. The lemma states that, if $\beta_m$ (or $\gamma_m$) is not empty or atomic, and does not have its outermost parentheses, then the parenthesis pair shown in $(\beta_m) \gamma_m (or \beta_m \gamma_m)$ is superfluous. Thus, we safely insert any missing parentheses around $\beta$ and $\gamma$, then repeatedly apply the same process on them, till we end up with a fully parenthesized expression of $E$. 

14
The relation \(<\) has the following properties:

1. \(<\) is transitive.
2. For every pair of distinct elements \(n_i, n_j\), either \(n_i < n_j\), \(n_j < n_i\), or there is an upper bound \(n_m\), such that both \(n_i, n_j < n_m\).
3. The set \(\{n_1, n_2, \ldots, n_k\}\) has a \(<\)-greatest element \(n_m\), i.e. for all \(i \neq m\), \(n_i < n_m\).

Note that Part 3 of the claim shows the existence of a position \(n_m\), which satisfies both of Conditions (1) and (2) of the lemma.

Proof of the claim:

1. Transitivity follows easily from Condition (1) of the theorem.
2. Let \(i < j\). From Condition (2) of the theorem, we get that whenever \(*_i \in [B \cup U_L]\) or \(*_j \in [B \cup U_R]\), then either \(*_i <_* L *_i \) or \(*_j <_* R *_i\), i.e. \(n_i < n_j\) or \(n_j < n_i\). The remaining case is when \(*_i \in U_R\) and \(*_j \in U_L\). Note that \(*_i\) can only bind a subword on its left, and therefore could not bind \(*_j\). Likewise \(*_j\) could not bind \(*_i\). Thus, there must be some binary operator \(*_m\) which could bind both \(*_i\) and \(*_j\), and therefore must occur between them, i.e. \(n_i < n_m < n_j\). Using Condition (2) of the theorem again, we get that \(*_i <_* L *_m\) and \(*_j <_* R *_m\), i.e. both \(n_i, n_j < n_m\).
3. By induction on \(k\) and using Parts (1) and (2) of the claim.

This finishes the proof of the claim, the lemma, and the theorem.

6 The Structure of Complete Precedence Systems

Let us try to closely understand how complete systems look like. We start by making the following:

Blanket Hypothesis 6.1 Throughout this section we let \(S = [<_L, <_R]\) be a complete precedence system.

We start by the following:

Definition 6.1 For \(o, * \in C\), we write \(o \sim * \) iff \((o <_* L * L o \) or \(o <_* R * R o\)). In other words, \(o \sim * \) iff \(*\) and \(o\) associate with each other either to the left or to the right.

We immediately get:

Proposition 6.2 The relation \(\sim\) on \(C\) is an equivalence relation, i.e. \(\sim\) is reflexive, symmetric, and transitive. Moreover, for each equivalence class \([*]\), either \(o <_* L * L o\) for each \(o \in [*]\), or \(o <_* R * R o\) for each \(o \in [*]\). Thus, using graph terminology, we can say that each equivalence class is either colored yellow (i.e. contains a yellow clique) or colored red (contains a red clique).

Proof: Taking \(o = *\) in Part 2 of Theorem 5.2, we get that either \(o <_* L o\) or \(o <_* R o\). Thus, \(\sim\) is reflexive. Also, \(\sim\) is symmetric by definition.

For the transitivity, let \(o \sim * \sim \Box\). Assume that \(o <_* L * L o\) (the case of \(o <_* R * R o\) being similar). If \( * <_* R \Box <_* R *\), then we get the yellow-red cycle \(* <_* L o <_* L o <_* R \Box <_* R *\), contradicting the unambiguity of \(S\). Thus, we must have \( * <_* L \Box <_* L *\). Now use the
transitivity of \( \prec_L \) (Part 2 of Theorem 5.2), to get that \( o \prec_L \square \prec_L o \), which shows that \( \sim \) is transitive.

Observe that, from the proof of transitivity, if \( o \sim * \sim \), then either
\[
o \prec_L * \prec_L \square \prec_L o \quad \text{or} \quad o \prec_R \prec_R \square \prec_R o.
\]
Thus, each equivalence class must have a uniform color. \( \Box \)

How do those cliques relate to each other? It turns out that they do that in a uniform way, as shown below:

**Proposition 6.3** Let \( o, o', *, *' \in C \) such that \( o \sim o' \) and \( * \sim *' \). Then:
\[
(o \prec_L * \text{ or } o \prec_R *) \iff (o' \prec_L *' \text{ or } o' \prec_R *')
\]

**Proof:** Of course, using symmetry only one direction suffices. Let us assume that \( o \prec_L * \) (the case of \( o \prec_R * \) being similar).

If \( *' \prec_R o' \), then we can easily check that there must be a yellow-red cycle connecting the 4 operators \( o, *, *', o' \), which would contradict the unambiguity of \( S \). Thus, \( *' \not\prec_R o' \).

If \( o' \in [B \cup U_L] \) or \( *' \in [B \cup U_R] \), then we can use Part 2 of Theorem 5.2 to get \( o' \prec_L *' \), and we are done.

Otherwise, \( o' \in U_R \) and \( *' \in U_L \). In this case, we must have \( o \prec_L o' \prec_L o \) and \( * \prec_R *' \prec_R * \), as this is the only way for \( o \sim o' \) and \( * \sim *' \) to hold.

Now, if \( *' \prec_R o' \), then we get the yellow-red cycle \( *' \prec_L o' \prec_L o \prec_L *' \), which would then contradict the unambiguity of \( S \). Thus, \( *' \not\prec_R o' \). We then apply Part 2 of Theorem 5.2 to get that \( o' \not\prec_R *' \), finishing the proof. \( \Box \)

The above proof reveals that on \( B \) we can actually get the stronger statements:

\[
o \prec_L * \iff o' \prec_L *'
\]
and
\[
o \prec_R * \iff o' \prec_R *
\]

The difficulty in getting that statement for unary operators appears when it is impossible to get \( o' \prec_L *' \) or \( o' \prec_R *' \), in case \( o' \) and \( *' \) are unary operators taking arguments from opposite sides. This is the reason behind the complexity of the statement in Proposition 6.3. In fact, many of the arguments in the next two propositions greatly simplify, had we been only interested in binary operators.

Proposition 6.3 was essential for justifying the following:

**Definition 6.2** On the set of equivalence classes \( C/\sim \), we define the relation:
\[
[0] \leq [s] \iff (o \prec_L * \text{ or } o \prec_R *).
\]

Note that Proposition 6.3 shows that \( \leq \) is a well defined relation on \( C/\sim \).

Moreover, we get:

**Proposition 6.4** \( (C/\sim, \leq) \) is a linear order, i.e. \( \leq \) is antisymmetric, transitive, and total.

**Proof:** For antisymmetry, suppose that \( [0] \leq [s] \leq [0] \). Then
\[
(o \prec_L * \text{ or } o \prec_R * \text{ or } o \prec_L \square \text{ or } o \prec_R \square).
\]
Since we must avoid yellow-red cycles, we must have \( o \prec_L * \prec_L o \) or \( o \prec_R * \prec_R o \). This shows that \( o \sim * \), i.e. \( [o] = [s] \).

For transitivity, suppose that \( [0] \leq [s] \leq [\square] \). Then
\[
(o \prec_L * \text{ or } o \prec_R *) \text{ and } (o \prec_L \square \text{ or } o \prec_R \square).
\]
Again we have 4 possibilities. Applying the transitivity of \( \prec_L \) and \( \prec_R \) (Part 1 of Theorem 5.2), two of the possibilities will lead to the required result. Assume that \( o \prec_L * \) but \(* \prec_R \Box \) (the 4th case being similar). In this case \( \Box \in (B \cup U_{L}) \). Since \( \Box \prec_L o \) will contradict the unambiguity of \( S \), we can use Part 2 of Theorem 5.2 to get that \( o \prec_R \Box \), i.e. \([o] \leq [\Box] \).

Finally, for the totality of \( \leq \), we need to show that \([o] \leq [\ast] \) or \([\ast] \leq [o] \), i.e.:

\[ o \prec_L * \text{ or } o \prec_R * \text{ or } * \prec_L o \text{ or } * \prec_R o . \]

But this easily follows from Part 2 of Theorem 5.2.

Proposition 6.4 shows that the class \( C \) of operators can be viewed as a tower of equivalence classes, where each floor (class) can be thought of being colored yellow for left associativity or red for right associativity.

We use the notation \([o] < [\ast] \) iff \([o] \leq [\ast] \) but \([o] \neq [\ast] \). The following theorem summarizes our findings:

**Theorem 6.5** Let \( o, \ast, \Box, \Diamond \in C \). then:

1. \([\Box] < [\Diamond] \) implies both \( \Box \prec_L \Diamond \) and \( \Box \prec_R \Diamond \), unless \( \Diamond \) is unary, in which case exactly one of the comparisons holds (the one that is allowed). Thus, operators living on lower floors precede those on higher floors in all possible ways.
2. If \( o, \ast \in U_{L} \), then \( o \prec_R \ast \prec_R o \). In particular, \([o] = [\ast] \). Thus, left unary operators live on a red floor, i.e. they associate with each other to the right.
3. If \( o, \ast \in U_{R} \), then \( o \prec_L \ast \prec_L o \). In particular, \([o] = [\ast] \). Thus, right unary operators live on a yellow floor, i.e. they associate with each other to the left.
4. If \( \Box \in (U_{L} \cup U_{R}) \) and \( \Box \in B \), then \([o] \leq [\Box] \). Thus, unary operators can not live above binary ones.
5. If \( \Box \in U_{L} \) and \( \ast \in U_{R} \), then either \([o] < [\ast] \) or \([\ast] < [o] \). This implies that either all \( U_{L} \) precede all \( U_{R} \) from the left, or all \( U_{L} \) precede all \( U_{R} \) from the right. Thus, from (4), \( U_{L} \) and \( U_{R} \) (if nonempty) occupy the first two floors, where some binary operators can share the second floor with unary ones.

**Proof:** For (1), if \([\Box] \leq [\Diamond] \), then one of \( \Box \prec_L \Diamond \) and \( \Box \prec_R \Diamond \) holds, say the former. Since \([\Box] \neq [\Diamond] \), we have \( \Diamond \neq \Box \). Using Part 2 of Theorem 5.2, we get \( \Box \prec_R \Diamond \), unless \( \Diamond \) is in \( U_{L} \).

(2), (3), and (4) follow from the particular cases of Part 2 in Theorem 5.2.

(5) This follows from the fact that we can not have \( o \sim \ast \), and thus left and right unary operators must live on different floors.

Letting \( L \) (\( R \)) be the set of left (right) associative operators, we get the following examples of natural complete precedence systems:

**Examples:**

1. Formulas of Propositional Logic;

\[ \{\neg\} \prec \{\land, \lor\} \prec \{\rightarrow\} \prec \{\leftrightarrow\} \]

\[ L = \{\land, \lor\}, R = \{\neg, \rightarrow, \leftrightarrow\} \]

2. Regular Expressions;

\[ \{\ast\} \prec \{\cdot\} \prec \{+\} \]

\[ L = \{+, \cdot, \ast\}, R = \emptyset \]
3. Arithmetic expressions;
\[\{\ominus, 1\} < \{\times, \div\} < \{+, -\}\]
\[\mathcal{L} = \{+,-,\times,\div\}, \mathcal{R} = \{\ominus, 1\}\]

We note here that the arithmetic expression \(e \uparrow \ominus x\) abbreviates \(e \uparrow (\ominus x)\), while \(\ominus e \uparrow x\) abbreviates \(\ominus (e \uparrow x)\), since \(\sim \ominus \) but both are right associative\(^6\).

The next example of an incomplete precedence system is due to an anonymous referee.

**Example:** In the programming language Modula-3, the infix binary operator “=” precedes the prefix unary operator “NOT”, violating Theorem 6.5, Item 4, and leaving the precedence system incomplete. Indeed, the parentheses in “a = (NOT b)” cannot be removed, and the quasi-expression “a = NOT b” is syntactically illegal.

Now let us stop and ask: Can there also be nullary operators (i.e. ones with no arguments)? Such operators behave like atomic expressions. In fact, we can view the atomic expressions as just operators with zero arity\(^7\).

If we want to extend the relations \(\prec_L\) and \(\prec_R\) to also cover nullary operators (i.e. atoms), then nullary operators precede other operators of positive arity, as the former ones will always be evaluated first.

Thus, let \(a\) be a nullary operator. If \(o \in (\mathcal{B} \cup \mathcal{U})\), then \(a \prec_R o\), and if \(o \in (\mathcal{B} \cup \mathcal{U})\), then \(a \prec_L o\). Thus, completing the picture, all nullary operators (atoms) live in the basement, which being under the ground, has no color.

This also gives an informal explanation of why, for complete precedence systems, unary operators may not be preceded by binary ones, namely, the former have fewer arguments to fight for.

**Acknowledgment**

The author would like to thank the two anonymous referees for their meticulous reading and helpful comments on the paper.

**References**


---

\(^6\)This is an even better reason for the right associativity of \(\uparrow\)

\(^7\)In predicate logic, logicians have adopted the convention that constant symbols are merely function symbols with zero arity, and propositional symbols are predicate symbols with zero arity.
We give here the algorithm of Proposition 3.8.

For $i = 1$ to $n$,
Let $C_L[i, 0] = U_R$, $C_R[i, 0] = U_L$.
Let $C_L[i, 1] = C_R[i, 1] = \varnothing$.
If $E[i, 1] \in \mathcal{A}$, then Let $C_L[i, 1] = B \cup U_L$, $C_R[i, 1] = B \cup U_R$.
Next i.
For $j = 2$ to $n$,
For $i = 1$ to $n - j + 1$,
Let $C_L[i, j] = C_R[i, j] = M[i, j] = \varnothing$.
If $E[i, 1] = "("$ and $E[i + j - 1, 1] = ")")$, then:
  If $M[i + 1, j - 2] \neq \varnothing$, then Let $C_L[i, j] = B \cup U_L$, $C_R[i, j] = B \cup U_R$.
Next i.
For $k = 1$ to $j$,
Let $\Delta = E[k, 1]$.
If $\Delta \not\in [C_R[i, k - 1] \cap C_L[i + k, j - k]]$, then Next k.
Let $M[i, j] = M[i, j] \cup \{k\}$.
For each $* \in \mathcal{C}$,
  If $\Delta \prec_L *$, then let $C_R[i, j] = C_R[i, j] \cup \{\ast\}$.
  If $\Delta \prec_R *$, then let $C_L[i, j] = C_L[i, j] \cup \{\ast\}$.
Next *.
Next k.
If $|M[i, j]| > 1$, then output “Ambiguous”.
Next i.
Next j.
If $|M[1, n]| = 0$, then output “Illegal Quasi-expression”.
If $|M[1, n]| > 0$, then output $M[1, n]$. //This is the set of positions of possible main operators.