BRACKET SYSTEMS

William Bricken

March 2002

Well-formed parentheses have been extensively studied, although usually as a trivial introduction to more useful systems, and never with the semantics of logic and circuits.

Kleene's 1952 *Introduction to Meta-Mathematics* [Kleene, p.23] provides some elementary theorems that one set of parentheses is sufficient to unambiguously fix the order of operations in a mathematical expression. His three lemmas proof that

1. Well-formed parentheses have an innermost pair,
2. There is only one proper pairing for well-formed parentheses, and
3. Subsets of well-formed parentheses are well-formed.

Languages composed solely of nested and concatenated parentheses are called both *Dyck languages* and *separator languages*. Thus, Kleene's properties are for Dyck languages. Separator languages permit any number of uniquely identified types of parentheses, but here we use only the simplest Dyck language with one type of parenthesis.

Perhaps the most relevant application of separator languages to date is to define the difference between context-free and context-sensitive languages in theoretical computer science [Davis, p.307]. The Chomsky-Schutzenberger representation theorem first constructs a language L consisting of a regular language combined with a separator language. L is context-free if and only if the separator language can be erased without creating an invalid word in L. This theorem can equivalently be stated that a language is context-free if all members of the alphabet of the regular language can be erased without creating a invalid word in the separator language. This theorem is important because it assures that parentheses can be added freely to establish operator precedence.

Separator languages are fundamental to the control of a pushdown automata, since the parentheses define which arguments go with which functions, and thus when a stack is needed to store temporary results. When reading a string, it is straight-forward for a compiler to interpret "(" as *push*, and ")" as *pop*, in the control of the stack.

Although these theorems and programming techniques of computer science were not derived with a logical interpretation in mind, they do however provide explicit information about how to implement boundary logic. We know, for instance, that a boundary logic form with no variables is both a circuit with bound inputs and a Dyke language. When the parens circuit is represented as a bit string, it can be processed in one pass, that is, without having to back-up the input, or process it again.

Encodings

Encoding "(" as 1 and ")" as 0 creates an appropriate bit-string, but is not optimal for bit-string processing. The production rules for such an encoding would be:

Meaning	*Parens form*	*Bit-string*
TRUE	( )	10
FALSE	void	empty
CALLING	( )( ) = ( )	1010 => 10
CROSSING	(( )) =	1100 => void

Another parsing technique comes directly from the LISP programming language. LISP uses parentheses exclusively to identify the order of evaluation and the assignment of arguments to functions. In the following encoding, parentheses are treated as binary trees.

PARENTHESIS TREES

( ) is valid, representing a leaf

(( )( )) is valid, representing a node with left and right branches

If P is valid, then so is (( ) P) and ((( )( )) P)

This encoding comfortably bridges the gap between graphs, logic, and integers [Jones]. The encoding of integers is:

Meaning	*Parens form*
0	( )
1	(( )( ))
2	(( ) (( )( )))
3	(( ) (( ) (( )( ))))

The conversion to bit-strings is

BIT-STRING PARENTHESIS TREES

Bracket token	*Bit-string token*
(	1
)	void
()	0

Thus the number 3 would have the bit-string encoding: 1010100.

Although the bit-string tree is not interpreted as logic, it is a small step to do so. The transformation rules below are expressed algebraically.

Meaning	*Parens form*	*Bit-string*		
TRUE	( )	0		
FALSE	void	void		
VOID OCCLUSION	(( ) A) =	10A	=>	void
INVOLUTION	((A)) = A	11A	=>	A
PERVASION	A (((A)) B) = A (B)	A111AB	=>	A1B

Above, Pervasion requires an Enfolding of the inner A since this encoding does not distinguish subforms sharing the same space. Thus, without ambiguity, we can express a circuit as a bit-string. For example, the simple ITE circuit, (((a) b)(((a)) c)) would encode as 111AB111AC, where inputs {a, b, c} would be bound to 0 if TRUE, and be erased if FALSE.

Another benefit of the computational use of separator languages, is that it gives us a direct measure of the complexity of processing parens strings as circuit simulations. Two approaches are possible. We could calculate the number of processor steps required for a stack-based implementation to process a bit-string representation of a circuit. Alternatively, we could use results from the implementation efficiency of LISP to compute the time required to process a parens tree.