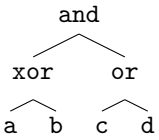
Boolean Algebra S-Expression Reducer (`Cimulink`)

Padraic Burns Jack Leightcap

Project Summary

An example of an S-Expression (Symbolic Expression) under Boolean Algebra might be `(and (xor a b) (or c d))`. The S-Expression grammar is comparatively easy to parse into an Abstract Syntax Tree (AST):



Once a raw string representing an S-Expression is parsed into an AST, tree traversals allow for the implementation of any arbitrary axioms. In this project, the AST is reduced (not "fully", but reasonably), and is able to be evaluated given input variables. This evaluation of 8 variables is mapped to the ZedBoard I/O, encompassing the features of Simulink's logic gates FPGA synthesis.

Project Description

This program largely takes its structure from basic compiler design [1].

Tokenizer (`token.c`)

The *tokenizer* splits an input string into *tokens*, smaller strings that are atoms of the intended parse. For an S-Expression, the atoms are

- lparen – `(`
- rparen – `)`
- operators – `and`, `or`, `xor`, and `not`
- variables – `0..7` and `T/F`

An expression `"(and 0 1)"` is then split into the tokens `"(", "and", "0", "1", ")"`. The choice to use numeric variable names was a direct design decision in simplifying the implementation of a tokenizer – the lack of character overlap with operators in addition to input validation with `atoi` simplified tokenizing greatly. This comes with the downside of initial ambiguity, where `0` and `1` are variables instead of booleans.

Parser (`parse.c`)

The *parser* parses tokens into a relevant data structure. Here, that structure is an Abstract Syntax Tree, leading to a naturally recursive parsing. The general procedure is:

1. If the current element is a left parenthesis, then the last element must be a right parenthesis, and the next element is an operator.
2. If the operator is `not`, then everything after the operator and before the right parenthesis is the singular argument. Return a tree with `not` as the node and recursively parse the argument as `arg0`.
3. If the operator is not `not`, then it has two branches. Split the elements after the operator and before the right parenthesis into two arguments. This is why parentheses are an important token, the point where the elements are split is the first point where the parentheses are balanced. Return a tree with the operator as the node and recursively parse the split elements as `arg0` and `arg1`.
4. Otherwise, the element is a variable (leaf of the tree). Return a tree with the variable as the node and two null branches.

Note that this is a combination of two atomic steps of compiler design, where some of this procedure would be considered *lexing* and others true *parsing*.

Evaluator (`eval.c`)

The *evaluator* inserts values for each variable into the tree, then recursively evaluates. The input array is simply mapped to each variable number, for example:

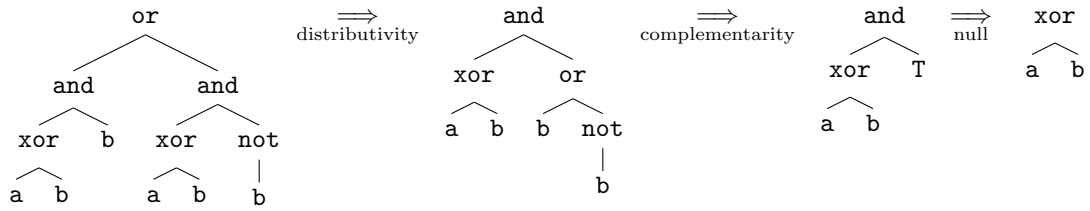
$$\underbrace{\{0,1,2,3,4,5,6,7\}}_{\text{vars}} := \underbrace{\{F,T,F,T,F,F,F,F\}}_{\text{args}}$$

The recursive evaluation is simply reading the node's operator, then applying that boolean function to the recursively evaluated arguments.

Reducer (`reduce.c`)

The *reducer* simply applies every axiom of Boolean Algebra at every node of the tree. This is clearly an expensive operation, so the more modular `tryall` is used to define a subset sequence of axioms to apply. For example, commutativity is an expensive axiom that switches every combination of every node's argument(s). This can be mitigated by simply making sure that the input S-Expression is right or left "justified".

An example reduction would be,



This can be verified using algebraic simplification,

$$\begin{aligned} f &= ((a \oplus b) \times b) + ((a \oplus b) \times b') \\ &= (a \oplus b) \times (b + b') \\ &= (a \oplus b) \end{aligned}$$

And finally with `cimulink`,

```
$ ./cimulink "(or (and (xor 0 1) 1) (and (xor 0 1) (not 1)))"
(xor 0 1)
```

ZedBoard I/O (`zedboard.c`)

ZedBoard I/O is used by passing a second argument `zedboard`,

```
$ ./cimulink "S-EXPRESSION" zedboard
```

This implements the given S-Expression on the ZedBoard, meaning the output of the given S-Expression is displayed on an LED given input from the 8 switches. Evaluation is significantly less expensive than reduction, so `eval.c` is simply used on the unreduced AST for input read from the switches.

Learning Outcomes

This project's inspiration came from the jump in topics between C++ and Simulink components of this class. The C++ portion of this class managed individual bits in something like `/dev/mem` to interface with the ZedBoard I/O. The Simulink portion of this class then jumped scale to using higher-order "atoms" like counters and equality checkers. This completely skips over the implementation between individual bits and more complex digital circuits.

This project combines both of these components – the individual bit setting of the C++ programs, as well as the digital logic design of Simulink projects. The choice to use C in place of C++ was a result of the imperative nature of this problem. While C++ OOP might have more relevance to the syllabus, the entirely imperative approach to this program means that C++ features would be unnecessary.

References

[1] Appel, Andrew W., and Ginsburg, Maia. *Modern Compiler Implementation in C*. Cambridge University Press, 2010.