Symbolic Computation

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

LGA Discussion

1. What questions did you have on the reading? Can your group members answer, or you can ask me.
2. Define **symbolic computation** in your own words.
3. What structures in Racket would you find useful for symbolic computation?
4. Share what other applications you came up with for symbolic computation. Formulate some more with your group.

Symbolic Computation Defined

- Wikipedia considers symbolic computation to be simply *computer algebra*.
- While computer algebra is a form of symbolic computation, there are plenty of other applications.
  - Programming languages
  - Compilers
  - Artificial intelligence
  - ...

Lisp & Symbolic Computation

Lisp dialects have a **homoiconic syntax**: the code is data, and data is code. Lists being the structure of the language syntax, code can be manipulated just like lists.

- The concept of "quoting" is fairly unique to just Lisp.
- It leads to a natural way to manipulate and work on *code* in the language.
- **Key point:** we can manipulate code before it is evaluated!

John McCarthy (1958)

Recursive Functions of Symbolic Expressions and their Computation by Machine

- Today we will explore a practical application of symbolic computation in artificial intelligence.

Boolean Expressions as S-Expressions

To represent boolean expressions in Racket, we need to formalize an s-expression syntax for them:

Conjunction	$a \land b \land c \ldots$	`(and a b c ...)`
Disjunction	$a \lor b \lor c \ldots$	`(or a b c ...)`
Negation	$\neg a$	`(not a)`

Practice: convert to s-expression

1. $a \land (b \lor c \lor d) \land d$
2. $\neg a \land (a \lor \neg b) \land \neg(a \lor b)$

Conjunctive Normal Form

Note

Depending on your background, you may already know this. Bear with me while I explain it to everyone else.

A boolean expression is in **conjunctive normal form** (CNF) if and only if all of the below are true:

- It only contains conjunctions, disjunctions, and negations.
- Negations only contain a variable, not a conjunction or disjunction.
- Disjunctions only contain variables and negations.

Example:

$$(a \vee b \vee c) \wedge (\neg a \vee b)$$

Learning Group Activity

Come up with an expression in CNF (not the example), and one not in CNF.

Verifying CNF in Racket

```
(define/match (in-cnf? expr [level 'root])
  [((? symbol?) _) #t]
  [(`(not ,(? symbol?)) _) #t]
  [((list-rest 'or args) (or 'root 'and))
   (andmap (λ (x) (in-cnf? x 'or)) args)]
  [((list-rest 'and args) 'root)
   (andmap (λ (x) (in-cnf? x 'and)) args)]
  [(_ _) #f])
```

Conversion to CNF

We can convert any boolean expression composed of just conjunctions, disjunctions, and negations to CNF using the following mathematical properties:

- **Elimination of double-negation:** $\neg\neg a \rightarrow a$
- **DeMorgan's Law (Conjunction):** $\neg(a \wedge b) \rightarrow (\neg a \vee \neg b)$
- **DeMorgan's Law (Disjunction):** $\neg(a \vee b) \rightarrow (\neg a \wedge \neg b)$
- **Distributive Property:** $a \vee (b \wedge c) \rightarrow (a \vee b) \wedge (a \vee c)$

Practice: Convert to CNF

Convert each expression to CNF:

1. $\neg(a \wedge \neg b)$
2. $\neg((a \vee b) \wedge \neg(c \vee d))$
3. $\neg((a \vee b) \wedge (c \vee d))$

Racket: Convert to CNF

Here's the base structure we want our code to follow:

```
(define (boolean->cnf expr)
  (if (in-cnf? expr)
    expr
    (boolean->cnf
      (match expr
        ...)))) ;; cases for the conversions we know
```

Double Negation Pattern Match

```
[`(not (not ,e)) e]
```

Simplify and/or of single argument

```
[`(or ,e) e]
[`(and ,e) e]
```

DeMorgan's Law

- DeMorgan's Law for Conjunction

```
[`(not (and ,@(list-rest args)))
  `(or ,@(map (curry list 'not) args))]
```

- DeMorgan's Law for Disjunction

```
[`(not (or ,@(list-rest args)))
  `(and ,@(map (curry list 'not) args))]
```

Explosion of and/or with nested expression

- and in and simplification

```
[`(and ,@(list-no-order (list-rest 'and inside) outside ...))
  `(and ,@inside ,@outside)]
```

- or in or simplification

```
[`(or ,@(list-no-order (list-rest 'or inside) outside ...))
  `(or ,@inside ,@outside)]
```

Distributive Law

```
[`(or ,@(list-no-order (list-rest 'and and-args) args ...))
  `(or ,@(cdr args)
       (and ,@(map
                (λ (x) (list 'or (car args) x))
                and-args)))]
```

Recurse otherwise...

```
[(list-rest sym args)
 (cons sym (map boolean->cnf args))]
```

Putting it all together

```
> (boolean->cnf '(or (and a b) (and (not c) d) (and (not e) f)))
'(and (or (not c) a (not e))
      (or (not c) b (not e))
      (or d a (not e))
      (or d b (not e))
      (or (not c) a f)
      (or (not c) b f)
      (or d a f)
      (or d b f))
```

The **satisfiability problem**[1] in computer science asks:

> *Given a boolean expression, is there any set of assignments to the variables which results in the equation evaluating to true?*

For example:

- `(and a (not a))`: not satisfiable
- `(and a a)`: satisfiable

(you could imagine much larger inputs)

[1] If you've taken algorithms, you probably know that this problem is **NP-complete**

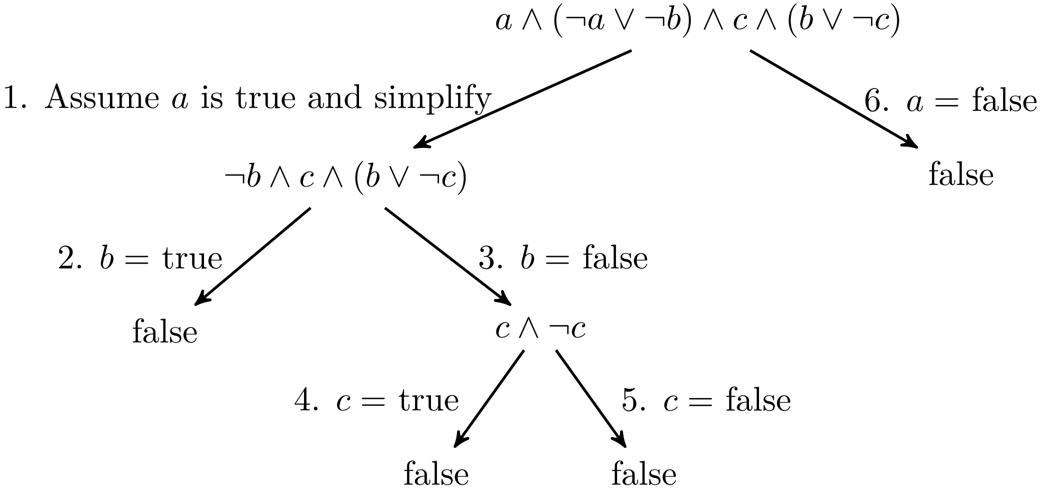
Davis-Puntam-Lodgemann-Loveland Algorithm

```
procedure DPLL(e):
   if e is true:
      return true
   if e is false:
      return false
   v ← select-variable(e)
   e₁ ← simplify(assume-true(v, e))
   if DPLL(e₁):
      return true
   e₂ ← simplify(assume-false(v, e))
   return DPLL(e₂)
```

Note

DPLL will work with any variable selection from `select-variable`, but certain selections may lead to a more efficent solution on average than others.

DPLL: Example



- We never reached true, so this equation is not satisfiable

DPLL: Exercise

Draw the DPLL tree for the following expression, and determine whether the equation is satisfiable or not:

$$(a \lor \neg b) \land (\neg a \lor b) \land (\neg a \lor \neg b)$$

DPLL in Racket

```
(define (solve-cnf expr)
  (define (solve-rec expr bindings)
    (case expr
      [(#t) bindings]
      [(#f) #f]
      [else
        (let ([sym (choose-symbol expr)])
          (define (solve-assume value)
            (solve-rec (assume sym value expr)
                       (cons (cons sym value) bindings)))
          (let ([sym-true (solve-assume #t)])
            (if sym-true
              sym-true
              (solve-assume #f))))]))
  (solve-rec expr '()))
```

choose-symbol

Not a good heuristic, but it works!

```
(define (choose-symbol expr)
  (if (symbol? expr)
    expr
    (choose-symbol (cadr expr))))
```

Assuming and Simplifying

```
(define (assume var value expr)
  (cond
    [(eq? var expr) value]
    [(equal? `(not ,var) expr) (not value)]
    [(symbol? expr) expr]
    [else
      (match expr
        [`(not ,_) expr]
        [(list-rest sym args)
         ...])])) ;; handle conjunction/disjunction
```

Handling Conjunction/Disjunction

```
(let ([look-for (case sym
                  [(and) #f]
                  [(or) #t])])
  (define (f item acc)
    (if (eq? acc look-for)
      acc
      (let ([result (assume var value item)])
        (cond
          [(eq? result look-for) result]
          [(eq? result (not look-for)) acc]
          [else (cons result acc)]))))
  (let ([result (foldl f '() args)])
    (cond
      [(null? result) (not look-for)]
      [(eq? result look-for) result]
      [else (cons sym result)])))
```

Putting It All Together

```
(define (solve expr)
  (solve-cnf (boolean->cnf expr)))

> (solve '(and a b))
'((b . #t) (a . #t))
> (solve '(or (and a b) (and c d) (and e f)))
'((d . #t) (f . #t) (c . #t))
> (solve '(and a (not a)))
#f
> (solve '(and (or (not a) b) (or a (not b))))
'((b . #t) (a . #t))
```