ECE570 Lecture 5: Symbolic Manipulation

Jeffrey Mark Siskind

School of Electrical and Computer Engineering

Fall 2013

PURDUE UNIVERSITY

A Simplifier for Arithmetic Expressions—I

```
(define (simplify e)
 (cond ((symbol? e) e)
       ((list? e)
        (case (first e)
          ((+) (simplify-+ e))
          ((-) (simplify-- e))
          ((*) (simplify-* e))
          ((/) (simplify-/ e))
          ((expt) (simplify-expt e))
          ((sqrt) (simplify-sqrt e))))))
```

A Simplifier for Arithmetic Expressions—II

```
(define (simplify-+ e)
 (case (length e)
  ((1) 0)
  ((2) (simplify (second e)))
  ((3) (list '+ (simplify (second e)) (simplify (third e))))
  (else
   (simplify (list '+ (second e) (cons '+ (rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—III

```
(define (simplify-- e)
 (case (length e)
  ((2) (simplify (list '* -1 (second e))))
  ((3) (simplify (list '+ (second e) (list '- (third e)))))
  (else
   (simplify (list '- (second e) (cons '+ (rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—IV

```
(define (simplify-* e)
 (case (length e)
  ((1) 1)
  ((2) (second e))
  ((3) (list '* (simplify (second e)) (simplify (third e))))
  (else
   (simplify (list '* (second e) (cons '* (rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—V

```
(define (simplify-/ e)
 (case (length e)
  ((2) (simplify (list 'expt (second e) -1)))
  ((3) (simplify (list '* (second e) (list '/ (third e)))))
  (else
   (simplify (list '/ (second e) (cons '* (rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—VI

```
(define (simplify-expt e)
 (list 'expt (simplify (second e)) (simplify (third e))))

(define (simplify-sqrt e)
 (simplify (list 'expt (second e) 0.5)))
```

A Symbolic Differentiator—I

```
(define (derivative e)
 (define (derivative e)
  (cond ((number? e) 0)
        ((symbol? e) (if (eq? e 'x) 1 0))
        ((list? e)
         (case (first e)
          ((+) (derivative-+ e))
          ((*) (derivative-* e))
          ((expt) (derivative-expt e))))))
 (simplify (derivative (simplify e))))
```

A Symbolic Differentiator—II

```
(define (derivative-+ e)
 (list '+ (derivative (second e)) (derivative (third e))))

(define (derivative-* e)
 (list '+
       (list '* (second e) (derivative (third e)))
       (list '* (third e) (derivative (second e)))))

(define (derivative-expt e)
 (list '*
       (third e)
       (list 'expt (second e) (- (third e ) 1))
       (derivative (second e))))
```

A Simplifier for Arithmetic Expressions—II(a)

```
(define (simplify-+ e)
 (case (length e)
  ((1) 0)
  ((2) (simplify (second e)))
  ((3) (let ((e1 (simplify (second e)))
             (e2 (simplify (third e))))
        (cond ((eqv? e1 0) e2)
              ((eqv? e2 0) e1)
              (else (list '+ e1 e2)))))
  (else
   (simplify (list '+ (second e) (cons '+ (rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—IV(a)

```
(define (simplify-* e)
 (case (length e)
  ((1) 1)
  ((2) (second e))
  ((3) (let ((e1 (simplify (second e)))
             (e2 (simplify (third e))))
        (cond ((eqv? e1 0) 0)
              ((eqv? e2 0) 0)
              ((eqv? e1 1) e2)
              ((eqv? e2 1) e1)
              (else (list '* e1 e2)))))
  (else
   (simplify (list '* (second e) (cons '* (rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—VI(a)

```
(define (simplify-expt e)
 (let ((e1 (simplify (second e)))
       (e2 (simplify (third e))))
  (if (eqv? e2 1) e1 (list 'expt e1 e2))))

(define (simplify-sqrt e)
 (simplify (list 'expt (second e) 0.5)))
```

Universal Quantification

```
(define (every procedure list)
 (or (null? list)
     (and (procedure (first list))
          (every procedure (rest list)))))
```

Existential Quantification

```
(define (some procedure list)
 (and (not (null? list))
      (or (procedure (first list))
          (some procedure (rest list)))))
```

Constant Folding

```
(define (constant? expression)
 (or (number? expression)
     (and (list? expression)
          (case (first expression)
           ((+) (every constant? (rest expression)))
           ((-) (every constant? (rest expression)))
           ((*) (every constant? (rest expression)))
           ((/) (every constant? (rest expression)))
           ((expt) (and (constant? (second expression))
                        (constant? (third expression))))
           ((sqrt) (constant? (second expression)))))))
```

A Simplifier for Arithmetic Expressions—I(b)

```
(define (simplify e)
 (cond ((constant? e) (calculate e))
       ((symbol? e) e)
       ((list? e)
        (case (first e)
         ((+) (simplify-+ e))
         ((-) (simplify-- e))
         ((*) (simplify-* e))
         ((/) (simplify-/ e))
         ((expt) (simplify-expt e))
         ((sqrt) (simplify-sqrt e))))))
```

A Programming Language with Derivatives

```
(define (evaluate expression definitions bindings)
 (cond
  ((symbol? expression) (lookup-variable expression bindings))
  ((list? expression)
   (case (first expression)
    ((+) ...)
    ((-) ...)
    ((*) ...)
    ((/) ...)
    ((expt) ...)
    ((sqrt) ...)
    ((if) ...)
    ((derivative)
     (evaluate
      (derivative (second expression)) definitions bindings))
    (else ...)))
  (else expression)))
```

A Simplifier for Arithmetic Expressions—II(c)

```
(define (simplify-+ e)
 (case (length e)
  ((1) 0)
  ((2) (simplify (second e)))
  ((3) (let ((e1 (simplify (second e)))
             (e2 (simplify (third e))))
        (cond ((eqv? e1 0) e2)
              ((eqv? e2 0) e1)
              (else `(+ ,e1 ,e2)))))
  (else (simplify `(+ ,(second e) (+ ,@(rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—III(c)

```
(define (simplify-- e)
 (case (length e)
  ((2) (simplify `(* -1 ,(second e))))
  ((3) (simplify `(+ ,(second e) (- ,(third e)))))
  (else (simplify `(- ,(second e) (+ ,@(rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—IV(c)

```
(define (simplify-* e)
 (case (length e)
  ((1) 1)
  ((2) (second e))
  ((3) (let ((e1 (simplify (second e)))
             (e2 (simplify (third e))))
        (cond ((eqv? e1 0) 0)
              ((eqv? e2 0) 0)
              ((eqv? e1 1) e2)
              ((eqv? e2 1) e1)
              (else `(* ,e1 ,e2)))))
  (else (simplify `(* ,(second e) (* ,@(rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—V(c)

```
(define (simplify-/ e)
 (case (length e)
  ((2) (simplify `(expt ,(second e) -1)))
  ((3) (simplify `(* ,(second e) (/ ,(third e)))))
  (else (simplify `(/ ,(second e) (* ,@(rest (rest e))))))))
```

A Simplifier for Arithmetic Expressions—VI(c)

```
(define (simplify-expt e)
 (let ((e1 (simplify (second e)))
       (e2 (simplify (third e))))
  (if (eqv? e2 1) e1 `(expt ,e1 ,e2))))

(define (simplify-sqrt e)
 (simplify `(expt ,(second e) 0.5)))
```

A Symbolic Differentiator—II(c)

```
(define (derivative-+ e)
 `(+ ,(derivative (second e)) ,(derivative (third e))))

(define (derivative-* e)
 `(+ (* ,(second e) ,(derivative (third e)))
     (* ,(third e) ,(derivative (second e)))))

(define (derivative-expt e)
 `(* ,(third e)
     (expt ,(second e) ,(- (third e ) 1))
     ,(derivative (second e))))
```