Brandeis University

Computer Science 21b

Structure and Interpretation of Computer Programs

Building a system for symbolic differentiation

Selectors, constructors and predicates:

`(constant?` e`)`	Is e a constant?
`(variable?` e`)`	Is e a variable?
`(same-variable?` v_1 v_2`)`	Are v_1 and v_2 the same variable?
`(sum?` e`)`	Is e a sum?
`(product?` e`)`	Is e a product?
`(addend` e`)`	Addend of the sum e.
`(augend` e`)`	Augend of the sum e.
`(multiplier` e`)`	Multiplier of the product e.
`(multiplicand` e`)`	Multiplicand of the product e.
`(make-constant` c`)`	Construct the constant c.
`(make-sum` a_1 a_2`)`	Construct the sum of a_1 and a_2.
`(make-product` m_1 m_2`)`	Construct product of m_1 and m_2.

Differentiation procedure:

```
(define (deriv exp var)
  (cond ((constant? exp) (make-constant 0))
        ((variable? exp)
         (if (same-variable? exp var)
             (make-constant 1)
             (make-constant 0)))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
         ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp))))))
```

Recall $\frac{d}{dx}x = 1$,$\frac{d}{dx}y = 0$, $\frac{d}{dx}(U+V) = \frac{d}{dx}U + \frac{d}{dx}V$, $\frac{d}{dx}(UV) = U\frac{d}{dx}V + V\frac{d}{dx}U$, and even

$$\frac{d}{dx}\frac{hi}{ho} = \frac{ho\frac{d}{dx}hi - hi\frac{d}{dx}ho}{hoho}.$$

Simple representations:

```
(define (constant? x) (number? x))
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2) (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (make-constant x) x)
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
(define (sum? x) (if (not (atom? x)) (eq? (car x) '+) nil))
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (product? x) (if (not (atom? x)) (eq? (car x) '*) nil))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
```

Extensions to the system

Simplification:

```
(define (make-sum a1 a2)
  (cond ((and (number? a1) (number? a2))
         (+ a1 a2))
        ((number? a1)
         (if (= a1 0)
             a2
             (list '+ a1 a2)))
        ((number? a2)
         (if (= a2 0)
             a1
             (list '+ a1 a2)))
        (else (list '+ a1 a2))))

(define (make-product m1 m2)
  (cond ((and (number? m1) (number? m2))
         (* m1 m2))
        ((number? m1)
         (cond ((= m1 0) (make-constant 0))
               ((= m1 1) m2)
               (else (list '* m1 m2))))
        ((number? m2)
         (cond ((= m2 0) (make-constant 0))
               ((= m2 1) m1)
               (else (list '* m1 m2))))
        (else (list '* m1 m2))))
```

Expanding to handle arbitrary numbers of arguments:

```
(define (augend s)
  (if (null? (cdddr s))
      (caddr s)
      (append '(+) (cddr s))))

(define (multiplicand p)
  (if (null? (cdddr p))
      (caddr p)
      (append '(*) (cddr p))))
```

Expanding to handle exponentiation: To the `deriv` procedure, add the following clause, and the subsequent procedures:

```
       ((exponential? exp)
        (make-product
         (make-product (exponent exp)
                       (make-exponential
                        (base exp)
                        (- (exponent exp) 1)))
         (deriv (base exp) var)))

(define (exponential? exp)
  (if (not (atom? exp)) (eq? (car exp) '**)))

(define (base exp) (cadr exp))

(define (exponent exp) (caddr exp))

(define (make-exponential b e)
  (cond ((= e 0) (make-constant 1))
        ((= e 1) b)
        (else (list '** b e))))
```

Pattern Matching and Simplification

The code on the following pages summarizes the pattern matcher that we will build in this lecture. The basic idea behind this simplification system is to build a set of rules, consisting of a pattern and an evaluation skeleton. Given an expression, we try to match the pattern to the expression, keeping track of the assignment of pattern variables to segments of the expression in a dictionary. If we can perform such a pattern match, then we use the skeleton part of the rule to indicate the procedures that should be evaluated to perform the simplification.

An example of the sort of rules we will wish to specify is the following set for algebraic simplification.

```
(define algebra-rules
 '(
   ( ((? op) (?c e1) (?c e2))         (: (op e1 e2))               )
   ( ((? op) (? e1) (?c e2))          ((: op) (: e2) (: e1))       )
   ( (+ 0 (? e))                      (: e)                        )
   ( (* 1 (? e))                      (: e)                        )
   ( (* 0 (? e))                      0                            )
   ( (* (?c e1) (* (?c e2) (? e3)))   (* (: (* e1 e2)) (: e3))     )
   ( (* (? e1) (* (?c e2) (? e3)))    (* (: e2) (* (: e1) (: e3))) )
   ( (* (* (? e1) (? e2)) (? e3))     (* (: e1) (* (: e2) (: e3))) )
   ( (+ (?c e1) (+ (?c e2) (? e3)))   (+ (: (+ e1 e2)) (: e3))     )
   ( (+ (? e1) (+ (?c e2) (? e3)))    (+ (: e2) (+ (: e1) (: e3))) )
   ( (+ (+ (? e1) (? e2)) (? e3))     (+ (: e1) (+ (: e2) (: e3))) )
   ( (+ (* (?c c) (? a)) (* (?c d) (? a)))
                                      (* (: (+ c d)) (: a))        )
   ( (* (? c) (+ (? d) (? e)))        (+ (* (: c) (: d))
                                         (* (: c) (: e)))          )
    ))
```

`(? x)` acts like a *variable* that can be bound to any expression;
`(?c x)` can similarly be bound to any numeric *constant*. Then
`(: x)` is the expression that was bound to `x`, and
`(: (op x y))` calls the Scheme evaluator (underneath!) with the expressions substituted in for `op`, `x`, `y`.

A simple matcher is given below, built on the appropriate data abstractions. This will be used to match the left side of rules against data expressions.

```
(define (match pattern expression dictionary)
  (cond ((eq? dictionary 'failed) 'failed)
        ((atom? pattern)
         (if (atom? expression)
             (if (eq? pattern expression)
                 dictionary
                 'failed)
             'failed))
        ((arbitrary-constant? pattern)
         (if (constant? expression)
             (extend-dictionary pattern expression dictionary)
             'failed))
        ((arbitrary-variable? pattern)
         (if (variable? expression)
             (extend-dictionary pattern expression dictionary)
             'failed))
        ((arbitrary-expression? pattern)
         (extend-dictionary pattern expression dictionary))
        ((atom? expression) 'failed)
        (else
         (match (cdr pattern)
                (cdr expression)
                (match (car pattern)
                       (car expression)
                       dictionary)))))
```

Once we have a consistent match, we instantiate the skeleton part of the rule, evaluating when appropriate.

```
(define (instantiate skeleton dictionary)
  (cond ((atom? skeleton) skeleton)
        ((skeleton-evaluation? skeleton)
         (evaluate (evaluation-expression skeleton)
                   dictionary))
        (else (cons (instantiate (car skeleton) dictionary)
                    (instantiate (cdr skeleton) dictionary)))))
```

The control of the rule base matcher consists of a series of tightly interwoven recursive loops. The basic idea is that given an expression, we apply the simplification rules in order until we either succeed or run out of rules. Moreover, if the expression to be simplified is compound, this control pattern is applied first to the elementary components of the expression, and then to the resulting simplified whole expression. The procedure `simplifier` on the next page returns a procedure, based on a set of rules, that can be applied to any expression.

```
(define (simplifier the-rules)
  (define (simplify-exp exp)
    (try-rules (if (compound? exp)
                   (simplify-parts exp)
                   exp)))
  (define (simplify-parts exp)
    (if (null? exp)
        '()
        (cons (simplify-exp (car exp))
              (simplify-parts (cdr exp)))))
  (define (try-rules exp)
    (define (scan rules)
      (if (null? rules)
          exp
          (let ((dictionary (match (pattern (car rules))
                                   exp
                                   (make-empty-dictionary))))
            (if (eq? dictionary 'failed)
                (scan (cdr rules))
                (simplify-exp (instantiate (skeleton (car rules))
                                           dictionary))))))
    (scan the-rules))
  simplify-exp)
```

Having built our simplification system, we now must build the data abstractions on which it rests. The dictionary is represented as a list of pairs, each pair consisting of the variable name and its associated value from the expression. Note that `assq` is a primitive Scheme procedure which has the behaviour of scanning a list of pairs, testing a key against the first element of each pair. If it finds a pair whose first element is equal to the key, then that pair is returned.

```
(define (make-empty-dictionary) '())

(define (extend-dictionary pat dat dictionary)
  (let ((vname (variable-name pat)))
    (let ((v (assq vname dictionary)))
      (cond ((eq? v #f)
             (cons (list vname dat) dictionary))
            ((equal? (cadr v) dat) dictionary)
            (else 'failed)))))

(define (lookup var dictionary)
  (let ((v (assq var dictionary)))
    (if (eq? v #f)
        var
        (cadr v))))
```

Predicates for types of expressions are straightforward.

```
(define (compound? exp) (pair? exp))
(define (constant? exp) (number? exp))
(define (variable? exp) (atom? exp))
```

A rule is represented by a list of two parts, a pattern and a skeleton.

```
(define (pattern rule) (car rule))
(define (skeleton rule) (cadr rule))
```

Within each pattern, explicit objects are represented by themselves, while generic objects are represented by a list, the first element of which identifies the type of generic object, and the second of which identifies the variable name of that object within the pattern. We distinguish three types of generic objects.

```
(define (arbitrary-constant? pat)
  (if (pair? pat) (eq? (car pat) '?c) #f))

(define (arbitrary-expression? pat)
  (if (pair? pat) (eq? (car pat) '?) #f))

(define (arbitrary-variable? pat)
  (if (pair? pat) (eq? (car pat) '?v) #f))

(define (variable-name pat) (cadr pat))
```

On the skeleton side of the rule, we have two procedures

```
(define (skeleton-evaluation? pat)
  (if (pair? pat) (eq? (car pat) ':) #f))

(define (evaluation-expression evaluation) (cadr evaluation))
```

The following procedure is included for completeness, but you need not worry about understanding exactly what it does until later in the term. For the purposes of this lecture, you can treat it as an abstraction that evaluates a given form, relative to the assignment of values to variables indicated in the dictionary.

```
(define (evaluate form dictionary)
  (write (list form dictionary)) (newline)
  (if (atom? form)
      (lookup form dictionary)
      (apply (eval (lookup (car form) dictionary))
             (map (lambda (v) (lookup v dictionary))
                  (cdr form)))))
```

Using the rules we called `algebra-rules` earlier we can define an algebraic simplification function as:

```
(define algsimp (simplifier algebra-rules))
```

Exactly the same system can be used to build other symbolic simplifiers. The following rules create a simple symbolic differentiator.

```
(define deriv-rules
 '(( (dd (?c c) (? v))              0                                  )
   ( (dd (?v v) (? v))              1                                  )
   ( (dd (?v u) (? v))              0                                  )
   ( (dd (+ (? x1) (? x2)) (? v))   (+ (dd (: x1) (: v))
                                       (dd (: x2) (: v)))             )
   ( (dd (* (? x1) (? x2)) (? v))   (+ (* (: x1) (dd (: x2) (: v)))
                                       (* (dd (: x1) (: v)) (: x2))) )
   ( (dd (** (? x) (?c n)) (? v))   (* (* (: n)
                                          (** (: x) (: (- n 1))))
                                       (dd (: x) (: v)))              )
    ))

(define dsimp (simplifier deriv-rules))
```

In closing, it is worth considering that important aspects of the kind of evaluation characterizing Scheme itself can be captured by such rules. This should not be entirely surprising, since these tables of rules define certain fixed kinds of substitutions, and it is the substitution model that is our current model of Scheme. For example:

```
(define scheme-rules
 '(( (square (?c n))              (: (* n n))                       )
   ( (fact 0)                     1                                 )
   ( (fact (?c n))                (* (: n) (fact (: (- n 1))))      )
   ( (fib 0)                      0                                 )
   ( (fib 1)                      1                                 )
   ( (fib (?c n))                 (+ (fib (: (- n 1)))
                                     (fib (: (- n 2))))             )
   ( ((? op) (?c e1) (?c e2))     (: (op e1 e2))                    ) ))

(define scheme-evaluator (simplifier scheme-rules))
```

A good exercise (though by no means trivial) is to work out the substitution model itself via such a set of rules. And for good measure, the computer psychiatrist is another example of symbolic rewriting through these kinds of replacement rules.

"Epistemology" is a big word from philosophy—in our case, what it means is "how do we represent knowledge?" In this handout, we see a *procedural* epistemology (represent what you know by programs that describe how) and a *data* epistemology (represent what you know by rules, with a piece of code that inteprets the rules correctly). But "correctly" is quite a condition—consider the following:

Suppose we have an algebra rule describing the commutativity of addition:

```
((+ (? a) (? b)) (+ (: b) (: a)))
```

The rules are applied to simplify an (algebraic) expression until none apply. So `(+ x y)` is "simplified" via the commutative law to `(+ y x)`... which is then simplified back to `(+ x y)`... and ...

Rules are also applied from the first to the last, in order. So if we invert the Scheme rules above, so we have instead

```
( (fact (?c n))                (* (: n) (fact (: (- n 1))))     )
( (fact 0)                     1                               )
```

then `(fact 0)` gets rewritten to `(* 0 (fact -1))`—no good.

So all of a sudden you realize: the *order* in which rules get applied affects the *result*. And the result may involve an infinite loop. So the rules are something which have to get *debugged*.

As a consequence, the rules *themselves* are no different from code written in your favorite programming language, for better and for worse. The rule interpreter (the matcher, instantiator, etc.) becomes a programming language interpreter. Data and programs end up being the same thing: when you write Scheme code, you think you're writing a *program*—but to the program that is the Scheme interpreter, your code is merely *data*.