VL Nichtprozedurale Programmierung

Funktionale Programmierung

Andreas Naderlinger
Fachbereich Computerwissenschaften
Software Systems Center
Universität Salzburg

UNIVERSITÄT SALZBURG

Klausurtermin

- 25.6.2018

Racket

(+ 3 5)

(fn arg arg …)

Parenthesis are never optional
They almost always mean: call function fn with those arguments

Java

```
public class MyFirstProgram {
  public static void main(String[] args) {

    int x = 3+5;
    System.out.println(x);      //we could have evaluated 3+5 here directly

  }
}
```

3 + 5	(+ 3 5)
sin(27)	(sin 27)
f(8,2)	(f 8 2)
10!	(fact 10)
$\sqrt{4}$	(sqrt 4)

The elements of programming

- How to combine simple ideas to form more complex ones?

- 3 mechanisms:
  – Primitive expressions
    e.g. 5 (number), “hello” (string), ‘aSymbol (symbol), + (procedure)
  – Means for combination
    e.g. (+ 5 3)   (…) procedure application
  – Means for abstraction
    e.g. (define pi 3.14159)
         (define radius 10)
         (define circumference (* 2 pi radius))

Evaluation

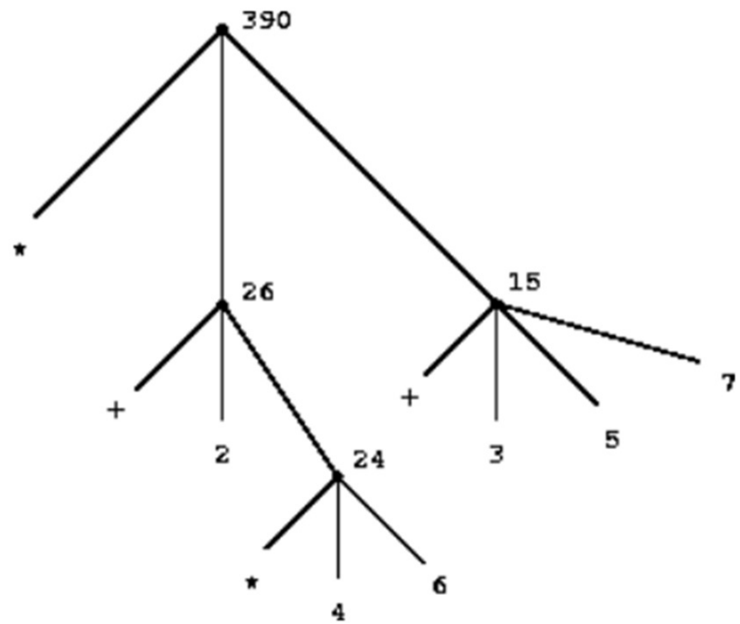
- (global) environment: keeps track of name-object pairs
- To evaluate a **combination**
  1. Evaluate the subexpressions of the combination
  2. Apply the procedure (leftmost subexpression i.e. operator) to the values of the other subexpressions (operands)

→ Recursive.

→ tree accumulation:

(* (+ 2 (* 4 6))
    (+ 3 5 7))

4 combinations



- To evaluate **primitive expressions**
  - The values of numerals are the numbers that they name
  - The values of other names are the objects associated with those names in the environment

    the values of built-in operators are the machine instruction sequences that carry out the corresponding operations

- Lisp has very simple syntax
  - A few **special forms**:

    e.g. define

    (define x 3) does not apply *define* to two arguments. → define is not a combination.

Compound Procedures

Demo1b

- A more powerful abstraction technique:

(define (square x) (* x x))

General form of a procedure definition:

(`define` (<*name*> <*formal parameters*>) <*body*>)

Evaluating the definition creates a compound procedure and associates it with a name (e.g. square).

Compound Procedures

Note: The body can be a sequence of expressions:

→ Evaluate each expression in sequence. Return the value of the final expression as the value of the procedure application

e.g. (define (square x) (+ x 1) x (* x x))

formal parameter

(define (square x) (* x x))

actual argument expression

(square (+ 2 3))

5 actual argument value

Examples & Substitution Model (1)

```
(square 21)
441

(square (+ 2 5))
49

(square (square 3))
81
```

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
25

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(f 5)
136
```

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

the basic principle:

```
(f 5)

(sum-of-squares (+ a 1) (* a 2))

(sum-of-squares (+ 5 1) (* 5 2))

(+ (square 6) (square 10))

(+ (* 6 6) (* 10 10))

(+ 36 100)

136
```

→ substitution model:
applicative-order evaluation

Racket uses applicative-order

Examples & Substitution Model (2)

```
(square 21)
441

(square (+ 2 5))
49

(square (square 3))
81
```

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
25

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(f 5)
136
```

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

an alternative model:

```
(f 5)

(sum-of-squares (+ 5 1) (* 5 2))

(+    (square (+ 5 1))      (square (* 5 2))  )

(+    (* (+ 5 1) (+ 5 1))   (* (* 5 2) (* 5 2)))
```

followed by the reductions

```
(+         (* 6 6)              (* 10 10))

(+           36                   100)

                   136
```

substitution model:
normal-order evaluation:
→ fully expand and then reduce

We want more ....

- The expressive power of the class of procedures that we can define so far is very limited.

Conditional Expressions & Predicates

Demo1c

- $$|x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

- cond : special form in Lisp for case analysis

$$
\begin{array}{l}
\texttt{(cond } (\langle p_1\rangle\ \langle e_1\rangle) \\
\qquad\quad (\langle p_2\rangle\ \langle e_2\rangle) \\
\qquad\quad \vdots \\
\qquad\quad (\langle p_n\rangle\ \langle e_n\rangle))
\end{array}
$$

paranthesized pairs of expressions: “clauses”

$<p_i>$ … predicate: expression that evaluates to true #t or false #f

Conditional Expressions & Predicates (2)

- $|x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

- cond else

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

- If

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

(for precisely two cases)

Conditional Expressions & Predicates (3)

- Primitive predicates <, =, >
- Logical composition operations
  - to construct compound predicates

```
(and <e1> ... <en>)

(or <e1> ... <en>)

(not <e>)
```

```
(and (> x 5) (< x 10))
```

Note: **`and`** and **`or`** are special forms (see later)

- >=

```
(define (>= x y)
  (or (> x y) (= x y)))
```

```
(define (>= x y)
  (not (< x y)))
```

How to determine the evaluation model

- applicative
- normal

Examples & Substitution Model (1)

```
(square 21)
441

(square (+ 2 5))
49

(square (square 3))
81
```

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
25

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(f 5)
136
```

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

the basic principle:

```
(f 5)

(sum-of-squares (+ a 1) (* a 2))

(sum-of-squares (+ 5 1) (* 5 2))

(+ (square 6) (square 10))

(+ (* 6 6) (* 10 10))

(+ 36 100)

136
```

→ substitution model:
applicative-order evaluation

Racket uses applicative-order

Examples & Substitution Model (2)

```
(square 21)
441

(square (+ 2 5))
49

(square (square 3))
81
```

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
25

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(f 5)
136
```

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

an alternative model:

```
(f 5)

(sum-of-squares (+ 5 1) (* 5 2))

(+    (square (+ 5 1))      (square (* 5 2))  )

(+    (* (+ 5 1) (+ 5 1))   (* (* 5 2) (* 5 2)))
```

followed by the reductions

```
(+         (* 6 6)             (* 10 10))

(+           36                   100)

                    136
```

substitution model:
normal-order evaluation:
→ fully expand and then reduce

How to determine the evaluation model

- applicative
- normal

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

Then he evaluates the expression

```
(test 0 (p))
```

Note: this is important! -- be sure to understand this.

Try also this: `(test 0 p)`

Recursion

- see *Fibonacci* (from lecture 0)
- see *evaluate a combination* (from this lecture)

- … a function calls itself as a subroutine
  – Direct vs. indirect recursion

- Efficiency
  – code size?
  – runtime?
  – memory?

```
(define (factorial n)
  (if (zero? n) 1
      (* n (factorial (sub1 n)))))
```

Special case (break condition)

- Widely used in functional programming
  – It helps to avoid mutation

[SICP, SimplyScheme]

Function vs. Procedure (1)

- A **function** is a connection between some values you already know and a new value you want to find out. (e.g. *square: 8 -> 64*)
- A **procedure** is a description of the process by which a computer can work out some result that we want.
- E.g. 2 definitions:
    - $f(x)=3x+12$
    - $g(x)=3(x+4)$
  - These two equations describe different *processes,* but they compute the same *function.* The function is just the association between the starting value(s) and the resulting value, no matter how that result is computed.
  - In Racket we'd say that f and g are two procedures that represent the same function.
    - (define (f x)   (+ (* 3 x) 12))
    - (define (g x)   (* 3 (+ x 4)))

[SICP, SimplyScheme]

Function vs. Procedure (2)

- An important difference between mathematical functions and computer procedures:
  - Procedures must be effective.
- E.g.

$$\sqrt{x} = \text{the y such that y } \geq 0 \; and \; y^2 = x$$

  - describes a perfectly legitimate mathematical function
  - we could use it to recognize whether one number is the square root of another or to derive facts about square roots in general.
  - does not describe a procedure
  - does not tell us how to actually find the square root of a given number.

iterative improvement

- $\sqrt[2]{x}$ *guess:* y *better guess:* $\dfrac{\left(\frac{x}{y}\right)+y}{2}$
- $\sqrt[3]{x}$ *guess:* y *better guess:* $\dfrac{\left(\frac{x}{y^2}\right)+2y}{3}$
- General computational strategy: **iterative improvement**

… to compute something,

- we start with an **initial guess** for the answer,
- test if the guess is **good enough**, and otherwise
- **improve** the guess and
- **continue** the process using the improved guess as the new guess.

Example: $\sqrt[2]{}$ by Newton's Method

- $\sqrt[2]{\boldsymbol{x}}$ *guess: y* *better guess:* $\dfrac{\left(\frac{x}{y}\right)+y}{2}$
- E.g. x = 2; initial guess y: 1

Guess	Quotient	Average
1	(2/1) = 2	((2 + 1)/2) = 1.5
1.5	(2/1.5) = 1.3333	((1.3333 + 1.5)/2) = 1.4167
1.4167	(2/1.4167) = 1.4118	((1.4167 + 1.4118)/2) = 1.4142
1.4142	. . .	. . .

1,41421356…

Formalize the process (with procedures)

- $\sqrt{x}$ → result (*sqrt*)
- first guess → result (*sqrtIter*)
  - guess → better guess (*improve*)
    - calculations (+,/)
    - stop? (goodEnough?)

Example: $\sqrt[2]{\quad}$ by Newton's Method (2)

Demo1d

```
(define (average x y)
  (/ (+ x y) 2))
```

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

```
(define (improve guess x)
  (average guess (/ x guess)))
```

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

No loops!

→ Sufficient for writing any purely numerical program that one could write in C, Java, etc.

Block structure and lexical scoping

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

Block structure and **lexical scoping**

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

References

-SICP
-Brian Harvey, Matthew Wright. Simply Scheme
-Robert W. Sebesta. Concept of Programming Languages, Pearson.
-Felleisen at al. Realm of Racket
-Ravi Chugh. Slides:UCSD: CSE 130 [Winter 2014] Programming Languages
-Neal Ford. Functional Thinking

Thank you!