CS131: Programming Languages

Fall 2015
Week #6

Today

- Prolog & HW 4 roundup
- Scheme Introduction
- HW 5
- Midterm

Prolog: fd_domain

```
fd_domain(Vars, Lower, Upper)
```

- Vars will be a list
  - More accurately, something unified to the form of a list
  - Lower & Upper are the inclusive bounds

```
fd_domain([X,Y], 1, 5).
   X = _#0(1..5)
   Y = _#17(1..5)


fd_domain(L, 1, 5). -- given L unified to a list
                       form
```

Prolog: fd_all_different

```
fd_all_different(Vars).
```

- Again, Vars will be a list of finite domain variables
- Constraint won't be triggered until one of the variables becomes a ground term

```
fd_domain([X,Y],1,5), fd_all_different([X,Y]).
   X = _#0(1..5)
   Y = _#17(1..5)
```

```
fd_domain([X,Y],1,5), fd_all_different([X,Y]), X=1.
   X = 1
   Y = _#17(2..5)
```

Prolog: fd_labeling

```
fd_labeling(Vars).
```

- Forces the finite domain vars to take on values

```
fd_domain([X,Y],1,2), fd_labeling([X,Y]).
   X = 1, Y = 1 ? ;

   X = 1, Y = 2 ? ;

   X = 2, Y = 1 ? ;

   X = 2, Y = 2

   yes
```

- Because Vars become ground terms, fd_all_different will be triggered!

HW 4: performance

- For performance reasons, use fd_labeling last
  - Whatever predicate you use fd_labeling in, order it last in the kenken predicate
- Also, only need to use fd_domain once on each row

Representing the KenKen board

`kenken(N,C,T)`

- T is a matrix - how do we represent?

Representing the KenKen board

`kenken(N,C,T)`

- T is a matrix - how do we represent?
  - List of lists
- But how do you force T to be of the form of a list of lists?

Representing the KenKen board

`kenken(N,C,T)`

- T is a matrix - how do we represent?
  - List of lists
- But how do you force T to be of the form of a list of lists?
  - Use the length predicate: `length(T,N)`
  - Also need to say every element of T is a list

Extracting/Using Constraint Info

```
+(5, [1-1, 2-1]),
-(3, 3-3, 4-3),
etc.
```

- These are forms - you can pattern match them

```
pred(+(N, L)) -- N = 5 and L = [1-1,2-1]
```

- 1-1, 2-1 are forms - you can pattern match them!
  - Need some way to index into T and get the finite domain variable
- Z #= X + Y
  - Use #= to do arithmetic when X, Y are fd variables

Scheme

- Functional language
- Popular Lisp dialect
- Minimal language specification
  - almost everything is a list

```
(function_id arg1 arg2 arg3 ...)

> (string-length “hello world”)
11
> (sqrt 16)
4
> (+ 1 2)
3
```

Scheme: Lists

```
(fun arg1 arg2 arg3 ...)
```

- Whatever identifier you put as the first element of the list, it will be interpreted as a function
  - So it better be one!
  - (1 2 3) - what will happen?

Scheme: Lists

```
(fun arg1 arg2 arg3 ...)
```

- Whatever identifier you put as the first element of the list, it will be interpreted as a function
  - So it better be one!
  - (1 2 3) - what will happen? error!
    - Trying to evaluate 1 as a function

Scheme: Lists

```
(fun arg1 arg2 arg3 ...)
```

- If you want a literal list, use quote

```
> (quote (1 2 3)) or ‘(1 2 3)
‘(1 2 3)
> (quote (+ 1 2))
‘(+ 1 2)
> (eval ‘(+ 1 2))
3
```

Scheme: Lambdas

- A lambda is an anonymous function
  - Lambda expression evaluates to a function

```
> (lambda (x) (+ x 2))
> (lambda (x y) (+ (+ x y) 2))
#<procedure>
```

- How do we call it?
  - Remember `(fun arg1 arg2 arg3 ...)`

Scheme: Lambdas

- A lambda is an anonymous function
  - Lambda expression evaluates to a function

```
> (lambda (x) (+ x 2))
> (lambda (x y) (+ (+ x y) 2))
#<procedure>
```

- How do we call it?
  - Remember `(fun arg1 arg2 arg3 ...)`

```
> ((lambda (x) (+ x 2)) 2)
4
```

Scheme: Define

- Use define to create a variable binding or a function

```
> (define x 4)
> x
4
> (define foo ________) -- what do we put here?
```

Scheme: Define

- Use define to create a variable binding or a function

```
> (define x 4)
> x
4
> (define foo (lambda (...) (...)))
> (define square _______)
```

Scheme: Define

- Use define to create a variable binding or a function

```
> (define x 4)
> x
4
> (define foo (lambda (...) (...)))
> (define square (lambda (x) (* x x)))
> (square 4)
```

- Equivalent syntax:

```
> (define (square x) (* x x))
```

Scheme: Control Flow

If statement

```
> (if (> 1 5)   “bigger” “smaller”)
“smaller”
```

```
> ((if #f + *) 5 3)
15
```

Cond

- Similar to a “switch-case” statement

```
> (define (reply-more s)
    (cond
        [ (equal? “hello” s) “hi!” ]
        [ (equal? “goodbye” s) “bye!” ]))
```

Scheme: Equivalence

- = is a function that compares integers
- equal? compares other forms

```
> (= 1 2)
#f
> (= ‘(1 2) ‘(1)) -- violation

> (equal? ‘(1 2) ‘(1))
#f
> (equal? ‘(+ 1 2) ‘(+ 1 2))
#t
> (equal? ‘(lambda (x) (+ x 1)) ‘(lambda (y) (+ y 1)))
#f
```

Scheme Practice: Factorial

```
(define factorial
```

Scheme Practice: Factorial

```
(define factorial
   (lambda (x)
      (if (= x 0) 1
         (* x (factorial (- x 1))))))

(define factorial
   (cond
      ((= x 0) 1)
      ((> x 0) (* x (factorial (- x 1))))))
```

Scheme Practice: Tail-recursive Factorial

```
(define (factorial x acc)
```

Scheme Practice: Tail-recursive Factorial

```
(define (factorial x acc)
   (if (= x 0) acc
      (factorial (- x 1) (* x acc))))
```

Scheme: List operations

- `(cons h t)` “Construct” - a function that returns a newly allocated list whose head is h and tail is t (given that t is a list)

- `(car p)` Returns the head of the list p
- `(cdr p)` Returns the tail of the list p

```
> empty
'()
> (cons “head” empty)
'(“head”)
> (cons “dead” (cons “head” empty))
'(“dead” “head”)
```

- Careful - t should be a tail. if not, cons will create a pair

```
> (cons 1 2)
'(1 . 2)  ; pair
```

Scheme: Named Let

- let allows you to do a local variable binding

```
> (let ((x 1) (y 2)) (+ x y))
3
```

- let* allows the bindings to refer to each other

```
> (let* ((x 1) (y x)) (+ x y))
2
```

Additional Scheme/Racket Resources

http://learnxinyminutes.com/docs/racket/

http://docs.racket-lang.org/guide/index.html

HW 5: Scheme Code Difference Analyzer

- `(compare-expr x y)`
  - it compares two Scheme expressions x and y
  - it produces a difference summary of where the two expressions are the same and where they differ

- examples

```
> (compare-expr 12 12)

12

> (compare-expr 12 20)

‘(if TCP 12 20)

> (compare-expr '(cons a b) '(cons a b))

‘(cons a b)
```

HW 5: Scheme Code Difference Analyzer

- More examples

```
> (compare-expr 'a '(cons a b))

(if TCP a (cons a b))

> (compare-expr '(if x y z) '(if x z z))

(if x (if TCP y z) z)
```

HW 5: Scheme subset

- Constant Literals
  - number, boolean (#t, #f), character, string
- Variable References
  - eg: x
- Procedure Calls
  - eg: (<operator> <operand1> <operand2> ...)
- The special forms:
  - (quote *datum*)
  - (lambda formals body)
  - (let bindings body)
  - (if expr expr expr)
  - (if expr expr)