6. Inductive Sets of Data

```
[[lecture 06 - Inductive Sets of Data.pdf]]
```

Inductive set definition

Defining set S

Top Down Definition - ends in base case

Definition 1.1.1: A natural number n is in S if and only if
1. $n = 0$, or
2. $n - 3 \in S$.

```
in-S?: N -> Bool
(define (in-S? n)
        (if (zero? n) #t
                (if (>= (- n 3) 0)
                        (in-S? (- n 3))
                )
        )
)
```

Bottom Up Definition - starts at base case

Definition 1.1.2: Define the set S to be the smallest set contained in N and satisfying the following two properties:

1. $0 \in S$, and
2. if $n \in S$, then $n + 3 \in S$

Question: Why is the "the smallest set" constraint needed?
Anwer: Because without this constraints, sets such as $S_2 = \{0, 3, 6, 9, 10\}$ are just as valid as $S_1 = \{0, 3, 6, 9\}$, since S_2 doesn't break any rules defined by the properties of the set.

Rules Of Inference Definition

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

Where,

- $0 \in S$ Axiom: The statement that is accepted True without proof.
- $n \in S$ Hypothesis (Antecedent): "if" part of the implication, which provides a condition.
- $(n+3) \in S$ Conclusion (Consequent): "Then" part of the implication, which follows from the hypothesis. If $n \in S$ then, $(n+3) \in S$.
- Syntax: Rules of inference follows a syntax based on division. If there is no divisor, the term is an axiom. And if there is a divisor, the term at the top is the hypothesis, and the divisor is the conclusion.

Defining a list of integers

Top Down Definition

A Scheme list is a list of integers if and only if,

1. It is an empty list, or
2. It is a pair whose car is an integer ans whose cdr is a list of integers.

Bottom Up Definition

1. $(.\,) \in \text{List-of-Int, and}$
2. $\text{if } n \in \text{Int and } ! \in \text{List-of-int, then } (n.\,1) \in \text{List-of-int}$

Rules Of Inference Definition

$$() \in \textit{List-of-Int}$$

$$\frac{n \in \textit{Int} \qquad l \in \textit{List-of-Int}}{(n\ .\ l) \in \textit{List-of-Int}}$$

Example:

- Show that (-7 3 14) is a list of integers:

`(-7 . (3 . (14 . ())))`

- Derivation (deduction tree)

$$\frac{\texttt{-7} \in N \qquad \dfrac{\texttt{3} \in N \qquad \dfrac{\texttt{14} \in N \qquad () \in \mathit{List\text{-}of\text{-}Int}}{\texttt{(14 . ())} \in \mathit{List\text{-}of\text{-}Int}}}{\texttt{(3 . (14 . ()))} \in \mathit{List\text{-}of\text{-}Int}}}{\texttt{(-7 . (3 . (14 . ())))} \in \mathit{List\text{-}of\text{-}Int}}$$

Lambda Calculus - Grammar

Defining S-list using grammar

$$\text{S-list} ::= (\{\text{S-exp}\}^*)$$
$$\text{S-exp} ::= \text{Symbol} \mid \text{S-list}$$

Examples:

$\text{S-list} \rightarrow ()$
$\text{S-exp} \rightarrow x$
$\text{S-list} \rightarrow (x)$
$\text{S-exp} \rightarrow (x)$
$\text{S-list} \rightarrow \text{((x) x (x) ((x) x (x))}

Syntax:

List-of-Int ::= ( )

$\underbrace{\text{List-of-Int}}_{\text{non-terminals}}$ ::= (Int . List-of-Int)

rows are productions.

is / can be

terminal

Kleene Notation:

- Star `{<exp>}*`: Indicates that the expression `<exp>` can be repeated zero or more times.
- Plus `{<exp>}+`: Indicates that the expression `<exp>` can be repeated one or more times.

- Separated List Plus `{<exp>}+{,}`: Indicates a list of one or more `<exp>` separated by commas.

Binary Tree

$$\text{Bintree} ::= \text{Int} \mid (\text{Symbol Bintree Bintree})$$

A binary tree in this example can either be an integer, or a symbol with two chil Binary trees. The symbol isn't explicitly defined, however, we can imagine it to consist of arithmetic operations, so that our binary tree in the end shows a treelike structure of calculations.

Proove that full binary trees have odd number of nodes

$$f(h) : \text{returns the amount of nodes in a full binary tree of height } h$$

Proof by induction:
Base Case: $f(1) = 1$, single node which is also the root of the tree
Inductive Step: $\text{if } f(h) \rightarrow f(h+1)$.
$f(h+1) = f(h) + 2^h$ Since $f(h)$ is odd, odd + even = odd

Lambda Expression

$$\begin{aligned} \text{LcExp} &::= \text{Identifier} \\ &::= (\text{lambda } (\text{Identifier})\ \text{LcExp}) \\ &::= (\text{LcExp LcExp}) \end{aligned}$$

Where an identifier is any symbol other than "lambda".

Examples:
$(\text{lambda } (x)\ x)$
$(\text{lambda } (x)\ (\text{lambda } (y)\ z))$

occurs-free?

If the given symbol affects the function from an outside scope, return True

```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```

- If the expression e is a variable, then the variable x occurs free in e if and only if x is the same as e.
- If the expression e is of the form `(lambda` (y) e'`)`, then the variable x occurs free in e if and only if y is different from x and x occurs free in e'.
- If the expression e is of the form $(e_1\ \ e_2)$, then x occurs free in e if and only if it occurs free in e_1 or e_2. Here, we use "or" to mean *inclusive or*, meaning that this includes the possibility that x occurs free in both e_1 and e_2. We will generally use "or" in this sense.

- The grammar

```
LcExp ::= Identifier
      ::= (lambda (Identifier) LcExp)
      ::= (LcExp LcExp)
```

- The procedure

```
occurs-free? : Sym × LcExp → Bool
usage:    returns #t if the symbol var occurs free
          in exp, otherwise returns #f.
(define occurs-free?
  (lambda (var exp)
    (cond
      ((symbol? exp) (eqv? var exp))
      ((eqv? (car exp) 'lambda)
       (and
         (not (eqv? var (car (cadr exp))))
         (occurs-free? var (caddr exp))))
      (else
        (or
          (occurs-free? var (car exp))
          (occurs-free? var (cadr exp)))))))
```

Let

```
(let (expressions) (body-to-eval))

(let
  ((x 3))
  (display x)
```

```
)
>> 3

(let
  ((x 3) (y 2))
  (display (+ x y))
)
>> 5
```

Important

In `let` we can't define variables based on other variables. We can use `letrec` for that.