Chapter 1: Inductive Sets of Data

- Recall definition of list
  - '() is a list.
  - If *l* is a list and *a* is any object, then **(cons** *a l***)** is a list
- More generally:
  - Define a simplest element
  - Define remaining elements by induction

Inductive Sets of Data

- Can define (natural) numbers this way:
  - 0 is a number
  - If *i* is a number, then (*s i*) is a number
- E.g., 3 is (*s* (*s* (*s* 0))) – “Church numerals”

- Russell & Whitehead (1925): try to build all of arithmethic like this (fails because of Gödel's Incompleteness theorem)

Backus-Nauer Form (BNF)

- Simplifies description of inductive data types

  <number> ::= 0, 1, 2, …

  <list-of-numbers> ::= ( )

  <list-of-numbers> ::= (<number> **.** <list-of-numbers>)

  [where (*a* . *b*) is like (*cons a b*)]

- Consists of

  Terminals: 0) **.** (

  Non-terminals: <list-of-numbers>

  Productions: <list-of-numbers> ::= ()

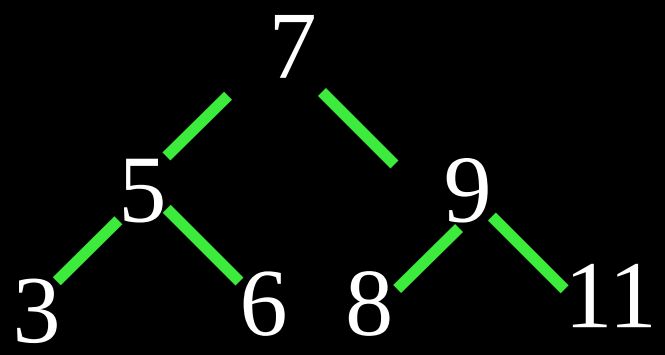
BNF

- Can only describe *context-free* structures; e.g.,

<bin-search-tree> ::=

(<key> . <bin-search-tree> <bin-search-tree>)

is inadequate for binary search trees:



which require a context-sensitive description

BNF: Notation

- *Disjunction* (this *or* that): |

  <number> ::= <even-number> | <odd-number>

- Kleene Star (zero or more): {}*

  <list-of-numbers> ::= ({<number>}*)

- Kleene Plus (one or more): {}+

  <nonempty-list-of-numbers> ::= ({<number>}+

Induction

What can we do with these (BNF) inductive definitions?

1) Prove theorems about data structures. E.g. given

<bintree> ::= <symbol> | (<bintree> <bintree>)

prove that a bintree always has an even # of parens

Induction

Basis: A tree derived from <bintree> ::= <symbol> has zero parens. Zero is even.

Inductive step: Assume that the relation holds for two bin trees b_1, with $2m \geq 0$ parens, and b_2, with $2n \geq 0$ parens. Using the rule <bintree> ::= (<bintree> <bintree>) we make a new bin tree $b_3 = (b_1\, b_2)$ from these, which has length $2m+2n+2 = 2[m+n+1]$, which is even. There is no other way to make a bin tree. Hence the relation holds for any bin tree with zero or more parens, Q.E.D.

Induction

What can we do with these (BNF) inductive definitions?

2) Write programs that manipulate inductive data

```
; count parentheses in a binary tree
(define count-bt-parens
        (lambda (bt)
          (if (atom? bt)
              0      ; base case
              (+ (count-bt-parens (car bt))
                 (count-bt-parens (cadr bt))
                 2))))  ; inductive step
```

Important Points

- Program mirrors BNF description mirrors proof.

```
(if (atom? bt)

    0
```

<bintree> ::= <symbol>

Basis: A tree derived from <bintree> ::= <symbol> has zero parens.

- Program mirrors BNF description mirrors proof:

```
(+ (count-bt-parens (car bt))
   (count-bt-parens (cadr bt))
   2))))
```

<bintree> ::= (<bintree> <bintree>)

Inductive step: Assume that the relation holds for two bin trees b_1, with $2m \geq 0$ parens, and b_2, with $2n \geq 0$ parens....

Follow the Grammar!

When defining a program based on structural induction, the structure of the program should be patterned after the structure of the data.

Another Point

- Program assumes we're passing it a binary tree: *dynamic type-checking* (vs. Java, *static* / compile-time type check)

- Easier to make type errors:

```
> (count-bt-parens '(dont taze me bro))
car: expects argument of type <pair>;
     given (dont taze me bro)
```

Another Example

```
; remove first occurrence of symbol from
; list of symbols
(define remove-first
  (lambda(s los)
          (if (null? los)
                   '()
                   (if (eqv? (car los) s)
                   (cdr los)
                   (cons (car los)
                            (remove-first
                                 s
                                 (cdr los)))))))
```

<list-of-symbols> ::= ( ) | (<symbol> . <list-of-symbols>)