ClojureBridge New York City

New York, NY

June 02 - 04, 2017

Basics

The List

Logic Theorist (1956)

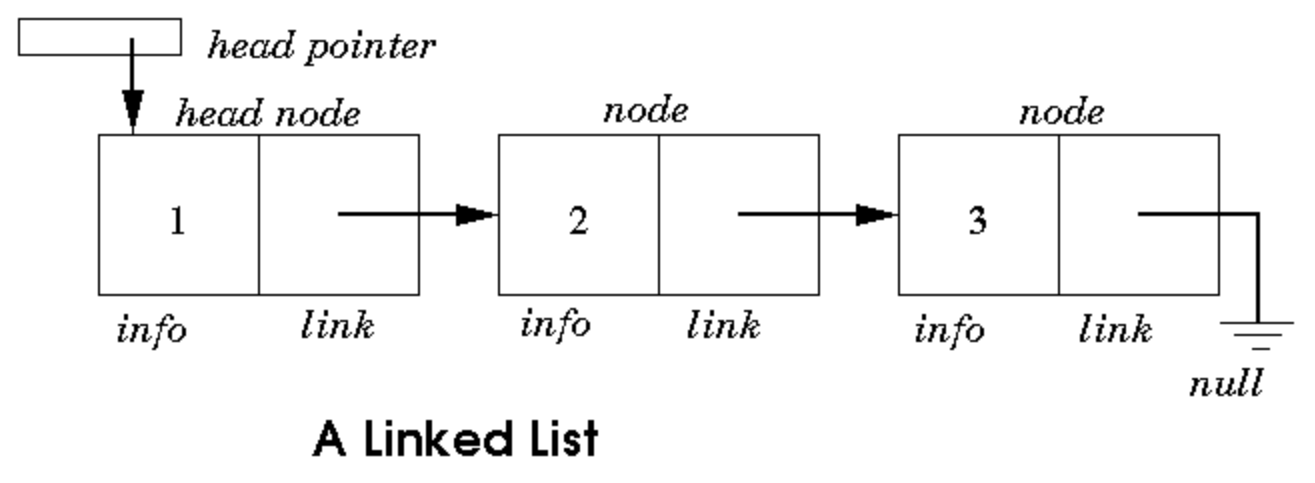
Allen Newell and Herbert Simon

Thinking Machine

Logic Theorist

- Generate mathematical proofs
  - Brute force
  - Heuristic solution
- Programmed by Cliff Shaw
- Discovery of the *linked list*
  - Flexible data structure
  - Recursive data structure

The Linked List

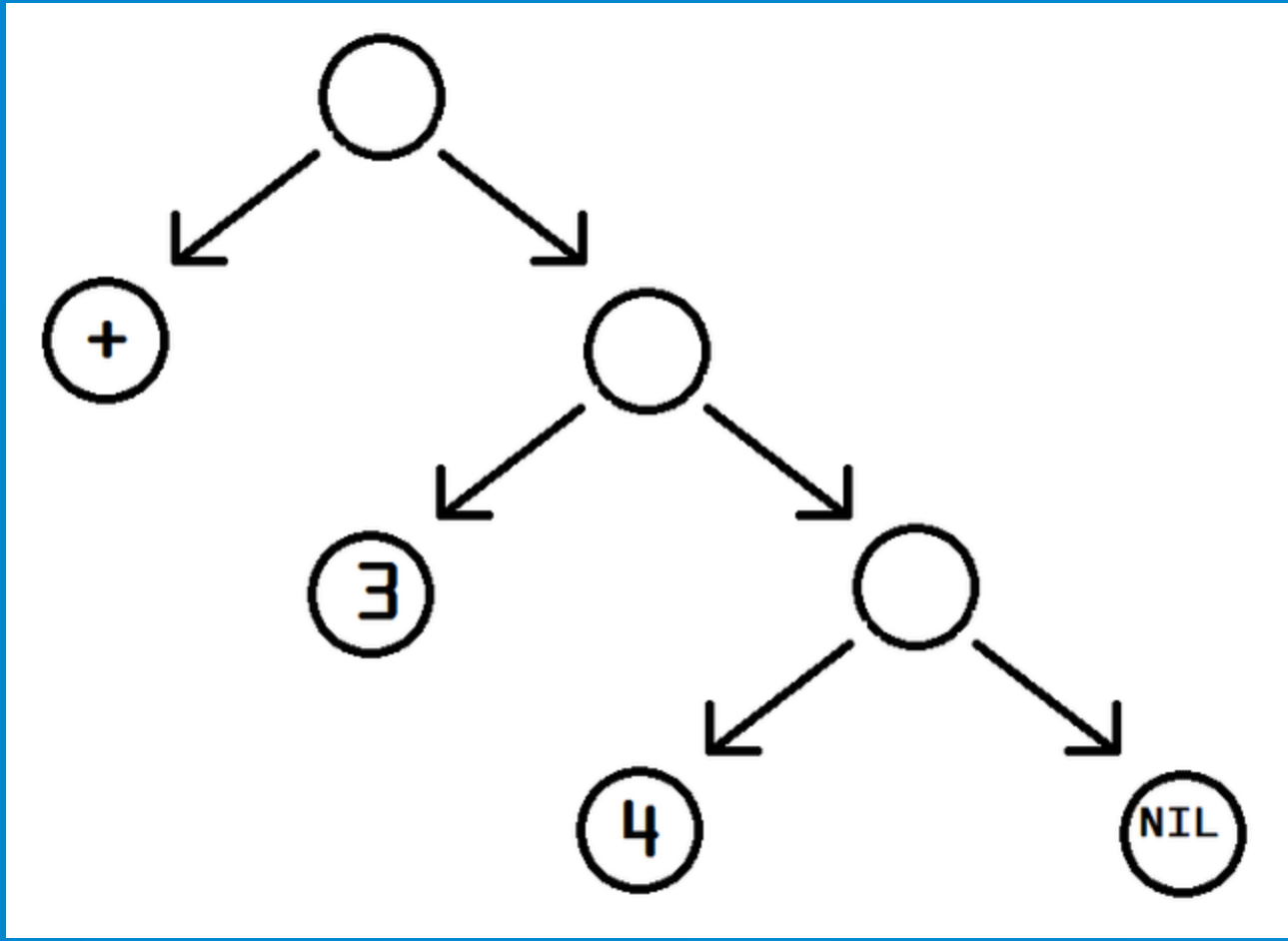


A Linked List

Logic Theorist

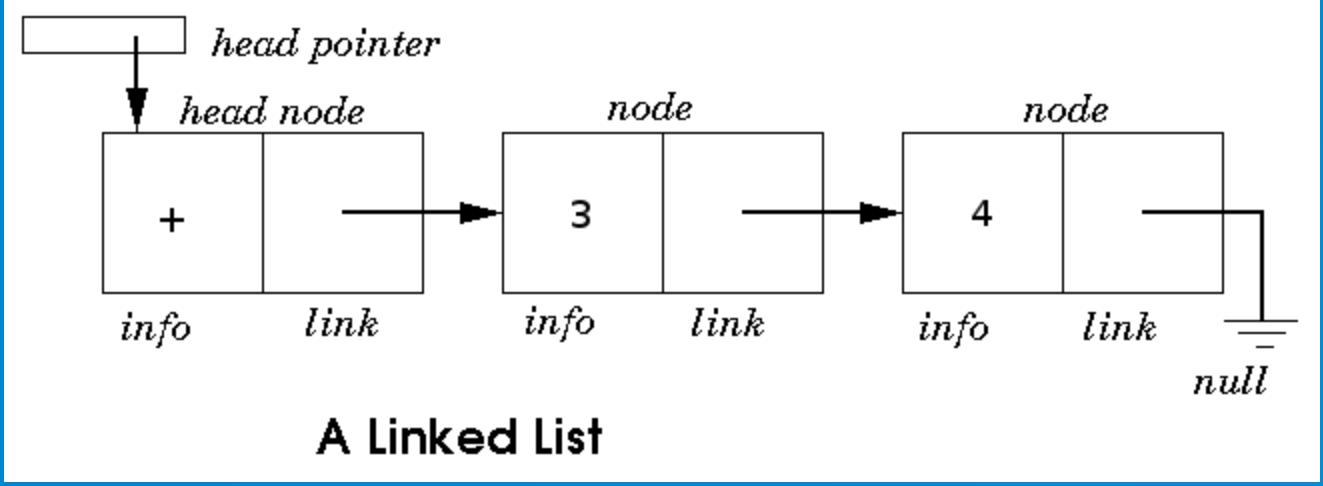
Proved 38 of the first 52 theorems in Whitehead and Russell's *Principia Mathematica*, and find new and more elegant proofs for some.

S-expressions



S-expressions

Represented as a List



A Linked List

S-expressions

Mathematical Expression - a combination of symbols that are well-formed according to syntactical rules. An arithmetical expression: `3 + 4`

Symbolic Expressions - a combination of symbols that are well-formed according to the structure of a nested binary tree. Directly expressed in first-order logic `(+ 3 4)`

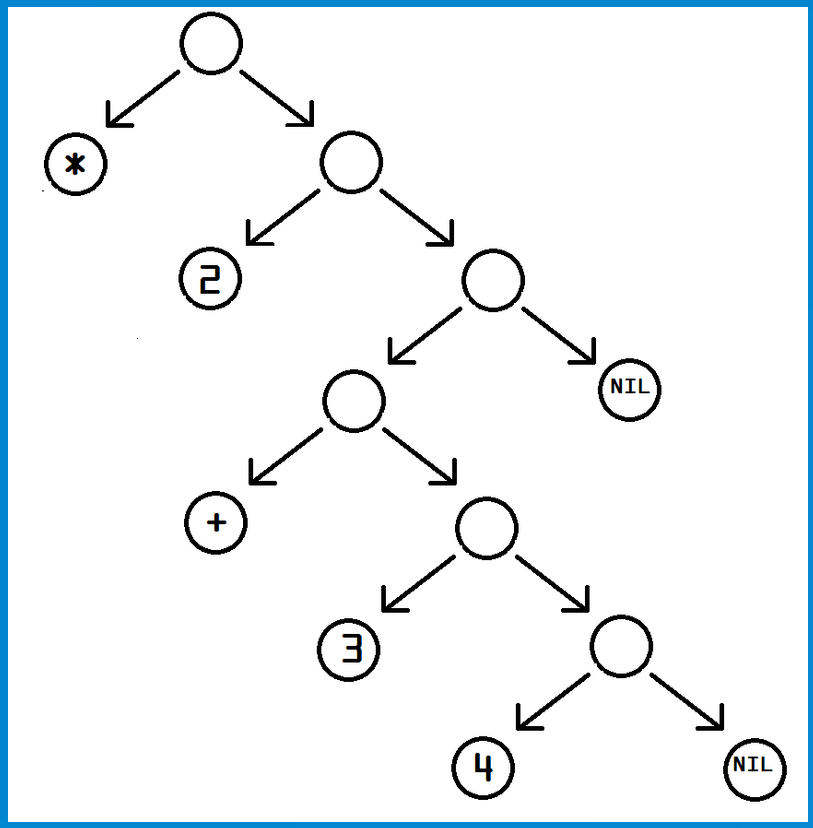
Clojure programs are composed of s-expressions.

S-expressions

```
(+ 3 4)
```

- Represents a *syntax tree* as a list. It contains three atoms: `+`, `3`, `4`.
- Represents a *form* when evaluated by Clojure
  - A *symbol* for an addition function called `+`
  - Two numeric literals, `3` and `4`

Prefix Notation



`(* 2 (+ 3 4))` **vs** `(2 * (3 + 4))`

S-expressions

Prefix Notation

(Those Parentheses)

- The parentheses ensure there is a unique parse tree for each expression.
- "Unique readability"
- `(predicate argument)`

S-expressions

Forms

Form - any s-expression meant to be evaluated

```
(+ 3 4)
> 7

(* 2 (+ 3 4))
> 14
```

Forms

Scalars: Numbers

Numeric literals are forms. They evaluate to themselves:

```
33
```

```
> 33
```

Forms

Scalars: Strings

A string is a form. It is denoted by quote marks, `" "`. They evaluate to themselves:

```
"Hello World"
> "Hello World"
```

Forms

Scalars: Symbols

Symbols are forms. They evaluate to what they name:

`inc` is a symbol that names a function

∴ Depending on the editor:

- `inc` ⇒ `#object[clojure.core/inc ...]`
- `inc` ⇒ `#function[clojure.core/inc]`

Forms

Documentation

- `doc` prints documentation for the form denoted by the symbol
- `(use 'clojure.repl)`
  - `(doc inc)` ⇒ `clojure.core/inc ...`
  - `(doc +)` ⇒ `clojure.core/+ ...`
  - `(doc 1)` ⇒ `ClassCastException java.lang.Long cannot be cast to clojure.lang.Symbol`

Forms

Documentation

Cheatsheet: https://clojure.org/api/cheatsheet

Forms

Scalars: Java Symbols

Symbols that begin with a dot to a Java class:

`(.toUpperCase "ClojureBridge")` ⇒ `"CLOJUREBRIDGE"`

Forms

Collections: Lists

A list is also a form. It is denoted by parentheses, `( )`. If the first element is a *symbol*, it's evaluated:

```
(inc 2)
> 3
```

```
(+)
> 0
```

```
(+ 2 1)
> 3
```

Forms and S-expressions Revisited

Form - any s-expression meant to be evaluated

- Scalar Form: `33` ⇒ `33`
- Collection Form: `(+ 3 4)` ⇒ `7`

Forms and S-expressions Revisited

Collection: `(1 2 3)`

- `(1 2 3)` ⇒ `ERROR`
  - This is data - a collection of numbers - not intended for evaluation
  - For example, there is no function called `1`

Forms and S-expressions Revisited

Collection: `(1 2 3)`

- `(list 1 2 3)` ⇒ `(1 2 3)`
  - This is code - a collection of symbols and literals - intended for evaluation
  - Calls `list` and returns a collection of numbers called a list

Forms and S-expressions Revisited

Unevaluated Form

```
(quote (+ 1 2))
> (+ 1 2)
```

```
'(a b c)
> (a b c)
```

S-expressions

Homoiconicity

S-expressions are used to represent **both** source **code** and **data**.

- A syntax tree is a **data** structure represented as a list, which can be evaluated as **code**
  - `(list '+ 1 2)` ⇒ `(+ 1 2)`
  - Evaluates to a list, which describes a function call

S-expressions

Homoiconicity

S-expressions are used to represent **both** source **code** and **data**.

- A form is **code** represented as a list, which can be evaluated to yield **data**
  - `(list 1 2 3)` ⇒ `(1 2 3)`
  - Evaluates to a list, which describes a collection

Boolean

Binary Predicates

- `(predicate argument)`
- Boolean forms
  - `(false? false)` ⇒ `true`
  - `(false? nil)` ⇒ `false`
  - `(nil? 0)` ⇒ `false`
  - `(true? ())` ⇒ `false`
  - `(zero? 0)` ⇒ `true`

Boolean

Special Form

- `(if test then else?)`
- Evaluates *test*

Special Form

Special forms are

a. Symbols
b. Only special when at the head of a list

Boolean

Conditionals

Branches based on the result of a form's evaluation

- `false` and `nil` are false
  - `(if nil "true" "false")` ⇒ `false`
  - `(if false "true" "false")` ⇒ `false`

Boolean

Conditionals

Branches based on the result of a form's evaluation

- `true` and everything else is true
  - `(if true "true" "false")` ⇒ `true`
  - `(if 88 "true" "false")` ⇒ `true`
  - `(if () "true" "false")` ⇒ `true`
- `(if (true? ()) "true" "false")` ⇒ `false`

Conditionals

- `(if test then else?)`
- `(cond & clauses)`
- `(condp pred expr & clauses)`
- `(case e & clauses)`

Conditionals

if

`=`, `>`, `>=`, `<`, `<=`, `==`, `not=`

```
(if (< 22 33)
    "true"
    "false")

> "true"
```

Conditionals

cond

- Forms as pairs
- Returns the first logical true

```
(cond
    (< -9 0) "negative"
    (> 9 0) "positive"
    :else "zero")

> "negative"
```

Conditionals

condp

Branches based on the result of a form's evaluation

- Binary predicate (`>`, `<`, `zero?`, etc...) and expression (scalar or collection)
- Test expression, result expression pairs
- Default expression

Conditionals

condp

```
(condp = 5
  1 "one"
  2 "two"
  3 "three"
  "too high")
```

Conditionals

case

```
(case (quote ())
  (()) "empty sequence"
  ((1 2)) "my sequence"
  "default")

> "empty sequence"

(case '(1 2)
  (()) "empty sequence"
  ((1 2)) "my sequence"
  "case not valid")

> "my sequence"
```