May the LISP be with You

Andrew (kqb) Easton

nerd2nerd: nerdend #cn4n

April 30, 2016

Thank You

- JK
- NaN

Our Example: Factorial

$n \in \mathbb{N}$ (alternatively: Natural n;)

$0! := 1$

$n! := n \cdot (n-1)!$

on the call stack:

$n! \rightarrow n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 2 \cdot 1$

Table of Contents

Runtime C/C++

```
1  int factorial(int n){
2          if (n == 0) {
3                  return 1;
4          }
5          else {
6                  return n * factorial(n - 1);
7          }
8  }
9
10 int fac4 = factorial(4);
```

Compiletime C

```
1 #define FAC0 1
2 #define FAC1 1
3 #define FAC2 2
4 #define FAC3 6
5 #define _FAC(X) FAC##X
6 #define FAC(X) _FAC(X)
7
8 int fac3 = FAC(3) // => 6
9 int fac4 = FAC(4) // => Error
```

Compiletime C++ (Function Templates)

```
1  template<int n> constexpr int fac() {
2          return n*fac<n-1>();
3  }
4
5  template<> constexpr int fac<0>() {
6          return 1;
7  }
8
9  constexpr int fac4 = fac<4>();
```

Table of Contents

Lisp Syntax

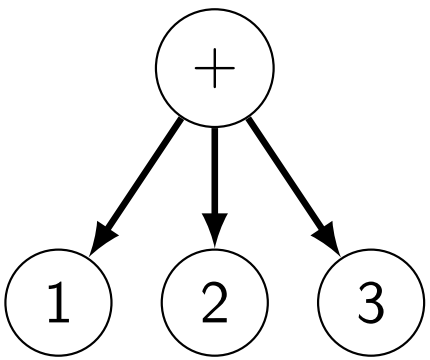
```
1 + 2 + 3;
```

```
h && i && k;
```

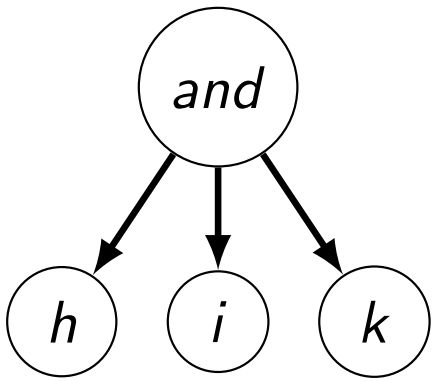
```
f(x, y);
```

Lisp Syntax

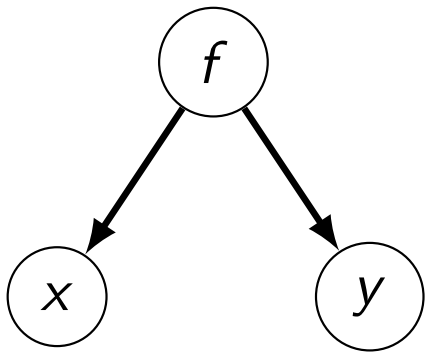
```
1 + 2 + 3;
```




```
h && i && k;
```

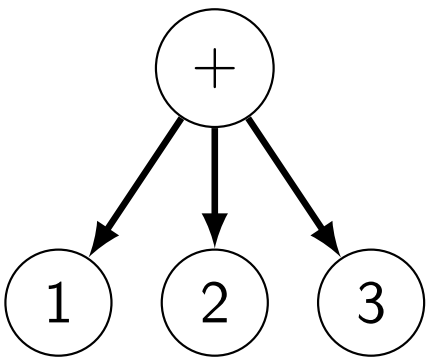



```
f(x, y);
```



Lisp Syntax

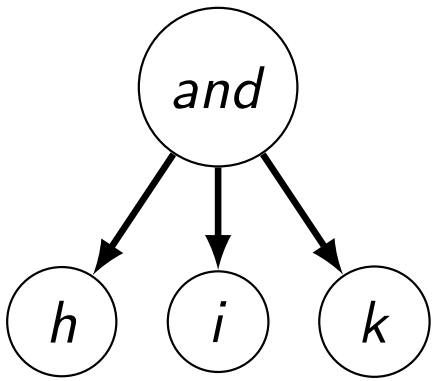
`1 + 2 + 3;`



`(+ 1 2 3)`

`+[1 2 3]`

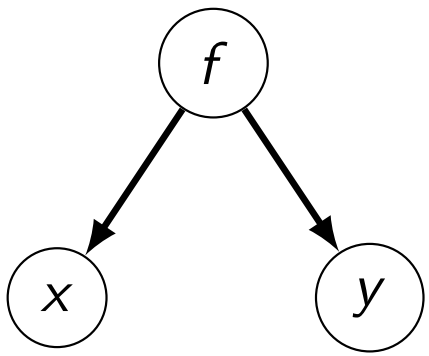
`h && i && k;`



`(and h i k)`

`and[h i k]`

`f(x, y);`



`(f x y)`

`f[x y]`

Lisp Syntax

```
1  (let [x 5]
2    x) ; => 5
3
4  (let [x 5]
5    (quote x)) ; => x
6
7  (let [x 5]
8    'x) ; => x
9
10 (let [x 5, y 6]
11   (list x y)); => (5 6)
12
13 (let [x 5, y 6]
14   (list 'x y)); => (x 6)
```

Runtime and Compiletime

```
1 (defn factorial [n]
2   (if (= n 0)
3     1
4     (* n (factorial (- n 1)))))
```

Runtime and Compiletime

```
1  (defn factorial [n]
2    (if (= n 0)
3      1
4      (* n (factorial (- n 1)))))

5  (defmacro ct-factorial [n]
6    (factorial n))
```

Runtime and Compiletime

```
1  (defn factorial [n]
2    (if (= n 0)
3      1
4      (* n (factorial (- n 1)))))

5  (defmacro ct-factorial [n]
6    (factorial n))

7  (factorial 4)
8  (ct-factorial 4)
```

Table of Contents

Lisp Template Syntax

```
1  (let [args (list x y)]
2
3    ;;; Aufruf
4    (f args) ; => (f (quote (x y)))
5
6    ;;; Templates / Schablonen
7    (backtick/template
8     (f args)) ; => (quote (f args))
9
10   (backtick/template
11    (f ~args)) ; => (quote (f (x y)))
12
13   (backtick/template
14    (f ~@args)) ; => (quote (f x y))
15
16   )
```

Table of Contents

An AST Pattern

```
1 (defn factorial [n]
2   (if (= n 0)
3     1
4     (* n (factorial (- n 1)))))

5 (defmacro ct-factorial [n]
6   (factorial n))
```

Onward to Macros

```
1 (defn factorial [n]
2   (if (= n 0)
3     1
4     (* n (factorial (- n 1)))))
```

Onward to Macros

```
1 (defn factorial [n]
2   (if (= n 0)
3     1
4     (* n (factorial (- n 1)))))
```

```
1 (backtick/template
2    (defn ~identifier ~arglist
3      ~@body))
```

Onward to Macros

```
1 (defmacro ct-factorial [n]
2   (factorial n))
```

Onward to Macros

```
1  (defmacro ct-factorial [n]
2    (factorial n))
```

```
1  (backtick/template
2    (defmacro ~(symbol (str "ct-" identifier)) ~arglist
3      (~identifier ~@arglist)))
```

Onward to Macros

```
1  ;; write once, use anywhere
2  (defmacro defrtct [identifier arglist & body]
3    (backtick/template
4     (do
5       (defn ~identifier ~arglist
6         ~@body)
7
8       (defmacro ~(symbol (str "ct-" identifier)) ~arglist
9         (~identifier ~@arglist)))))
```

Onward to Macros

```
1  ;; write once, use anywhere
2  (defmacro defrtct [identifier arglist & body]
3    (backtick/template
4     (do
5       (defn ~identifier ~arglist
6         ~@body)
7
8       (defmacro ~(symbol (str "ct-" identifier)) ~arglist
9         (~identifier ~@arglist)))))
```

```
1  (defrtct factorial [n]
2    (if (= n 0)
3      1
4      (* n (factorial (- n 1)))))
```

Onward to Macros

```
1  (defn factorial [n]
2    (if (= n 0)
3      1
4      (* n (factorial (- n 1)))))

5  (defmacro ct-factorial [n]
6    (factorial n))
```

Onward to Macros

```
1  (defn factorial [n]
2    (if (= n 0)
3      1
4      (* n (factorial (- n 1)))))

5  (defmacro ct-factorial [n]
6    (factorial n))

1  (defrtct factorial [n]
2    (if (= n 0)
3      1
4      (* n (factorial (- n 1)))))
```

Onward to Macros

```
1 (defrtct factorial [n]
2   (if (= n 0)
3     1
4     (* n (factorial (- n 1)))))
```

Onward to Macros

```
1 (defrtct factorial [n]
2   (if (= n 0)
3     1
4     (* n (factorial (- n 1)))))
```

Onward to Macros

```
1 (defrtct factorial [n]
2   (if (= n 0)
3     1
4     (* n (factorial (- n 1)))))
```

```
1 (factorial 4)
2 (ct-factorial 4)
```

Revised: Runtime and Compiletime

```
1 (defn rt-factorial [n]
2   (if (= n 0)
3     1
4     (* n (rt-factorial (- n 1)))))
```

Revised: Runtime and Compiletime

```
1  (defn rt-factorial [n]
2    (if (= n 0)
3      1
4      (* n (rt-factorial (- n 1)))))

5  (defmacro factorial [n]
6    (let [x (try (eval n) (catch Exception e nil))]
7      ;; 'number? ==> fail early on floating point
8      (if (number? x)
9        (rt-factorial x)
10       (backtick/template
11        (rt-factorial ~n)))))
12
13 ;; ==> write once, use anywhere, anytime
```

Table of Contents

To ∞ and Beyond: Learning Lisp for Free

- "Logic, Math, Structure"
  - Haskell B. Curry: U-Language
- Common Lisp
  - Peter Seibel: Practical Common Lisp
  - Edmund Weitz: Common Lisp Recipes (only $\frac{1}{2}$ free)
  - LispWorks: the Common Lisp HyperSpec
  - Toronto Lisp User Group: Learn Lisp the Hard Way
- Macro Programming (Common Lisp)
  1. Paul Graham: On Lisp
  2. Doug Hoyte: Let over Lambda (only $\frac{1}{2}$ free)
- Clojure
  - Daniel Higginbotham: Clojure for the Brave and True
  - *build tool*: leiningen, **boot-clj**

To ∞ and Beyond: Learning Lisp for Free

- "Logic, Math, Structure"
  - Haskell B. Curry: U-Language
- Common Lisp
  - Peter Seibel: Practical Common Lisp
  - Edmund Weitz: Common Lisp Recipes (only $\frac{1}{2}$ free)
  - LispWorks: the Common Lisp HyperSpec
  - Toronto Lisp User Group: Learn Lisp the Hard Way
- Macro Programming (Common Lisp)
  1. Paul Graham: On Lisp
  2. Doug Hoyte: Let over Lambda (only $\frac{1}{2}$ free)
- Clojure
  - Daniel Higginbotham: Clojure for the Brave and True
  - *build tool*: leiningen, **boot-clj**

To ∞ and Beyond: Learning Lisp for Free

- "Logic, Math, Structure"
  - Haskell B. Curry: U-Language
- Common Lisp
  - Peter Seibel: Practical Common Lisp
  - Edmund Weitz: Common Lisp Recipes (only $\frac{1}{2}$ free)
  - LispWorks: the Common Lisp HyperSpec
  - Toronto Lisp User Group: Learn Lisp the Hard Way
- Macro Programming (Common Lisp)
  1. Paul Graham: On Lisp
  2. Doug Hoyte: Let over Lambda (only $\frac{1}{2}$ free)
- Clojure
  - Daniel Higginbotham: Clojure for the Brave and True
  - *build tool*: leiningen, **boot-clj**

To ∞ and Beyond: Learning Lisp for Free

- "Logic, Math, Structure"
  - Haskell B. Curry: U-Language
- Common Lisp
  - Peter Seibel: Practical Common Lisp
  - Edmund Weitz: Common Lisp Recipes (only $\frac{1}{2}$ free)
  - LispWorks: the Common Lisp HyperSpec
  - Toronto Lisp User Group: Learn Lisp the Hard Way
- Macro Programming (Common Lisp)
  1. Paul Graham: On Lisp
  2. Doug Hoyte: Let over Lambda (only $\frac{1}{2}$ free)
- Clojure
  - Daniel Higginbotham: Clojure for the Brave and True
  - *build tool*: leiningen, **boot-clj**

To ∞ and Beyond: Learning Lisp for Free

- Worse is Better
- The Lisp Curse
- The Bipolar Lisp Programmer
  (offline since approx. 2016-02)