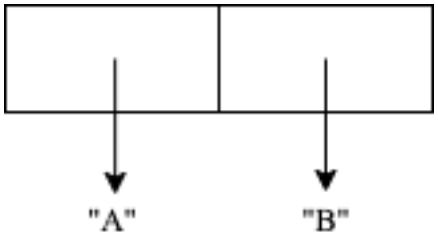
Overview

1. Cons Cell Basics
2. How The Lisp Compiler Compiles Lists
3. Compiler Basics
4. Programs That Write Programs
5. Macro Syntax (Quasiquote, Comma, At)

1. Cons Cell

1. Field Names

1. Cons Cell With Content

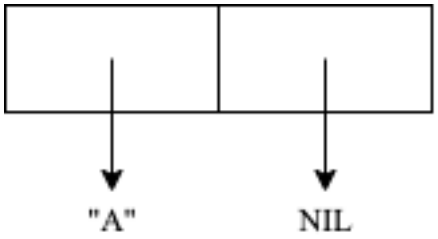


Written as

("A" . "B")

N.B. The DOT is the boundary between the CAR and the CDR

1. NIL



Written as

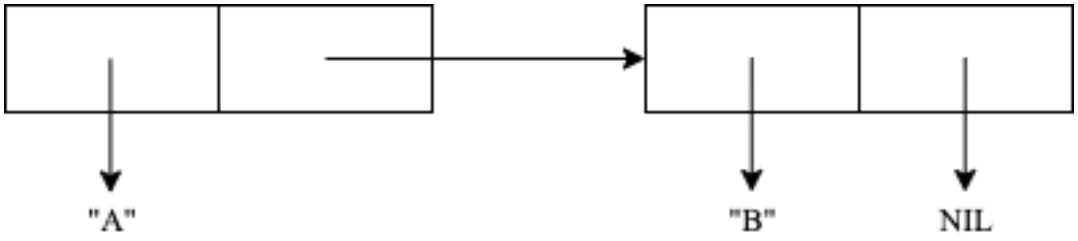
("A" . NIL)

Also written as

("A")

} Most common

1. List



Written as

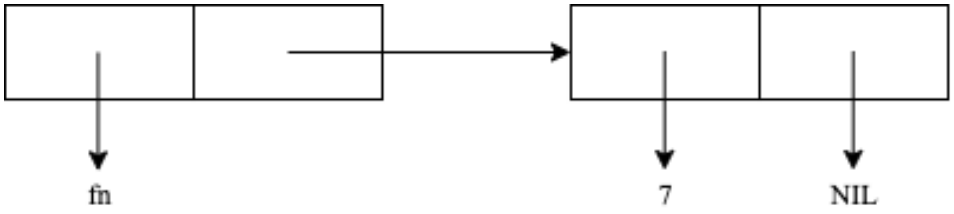
("A" . ("B" . NIL))

Also written as

("A" "B")

Most common

2. Function



Lisp compiler assumes that the 1st element of a list is a function

Also written as

(fn . (7 . NIL))

Written as

(fn 7)

Lisp compiler assumes that the rest of the elements of the list are function parameters

Other languages

fn(7)

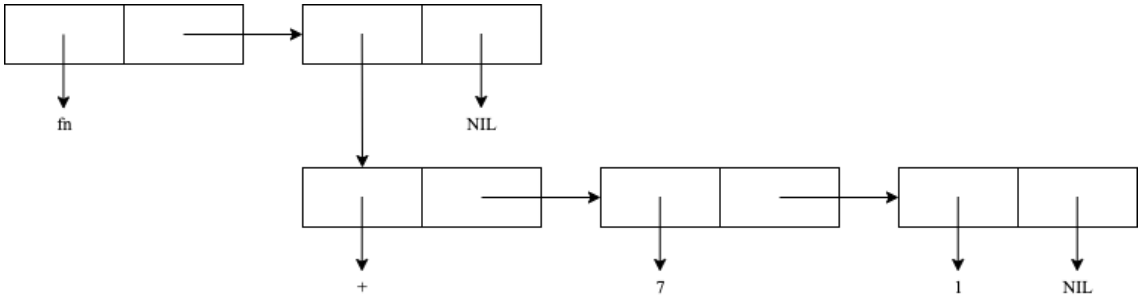
N.B same number of parens

N.B. Functions are lists

N.B. Lisp has many operators / library functions to operate on lists

N.B. Lisp allows various other things to be atoms (e.g. like the strings in previous examples) numbers, strings, identifiers (called Symbols), etc., etc.

2. Add (Function)



Also written as

(fn . (7 . (+ . (1 . NIL))))

Written as

(fn (+ 7 1))

Lisp compiler interprets 1st element of a list to be a function.

In this case, there are *two* first elements: fn and +

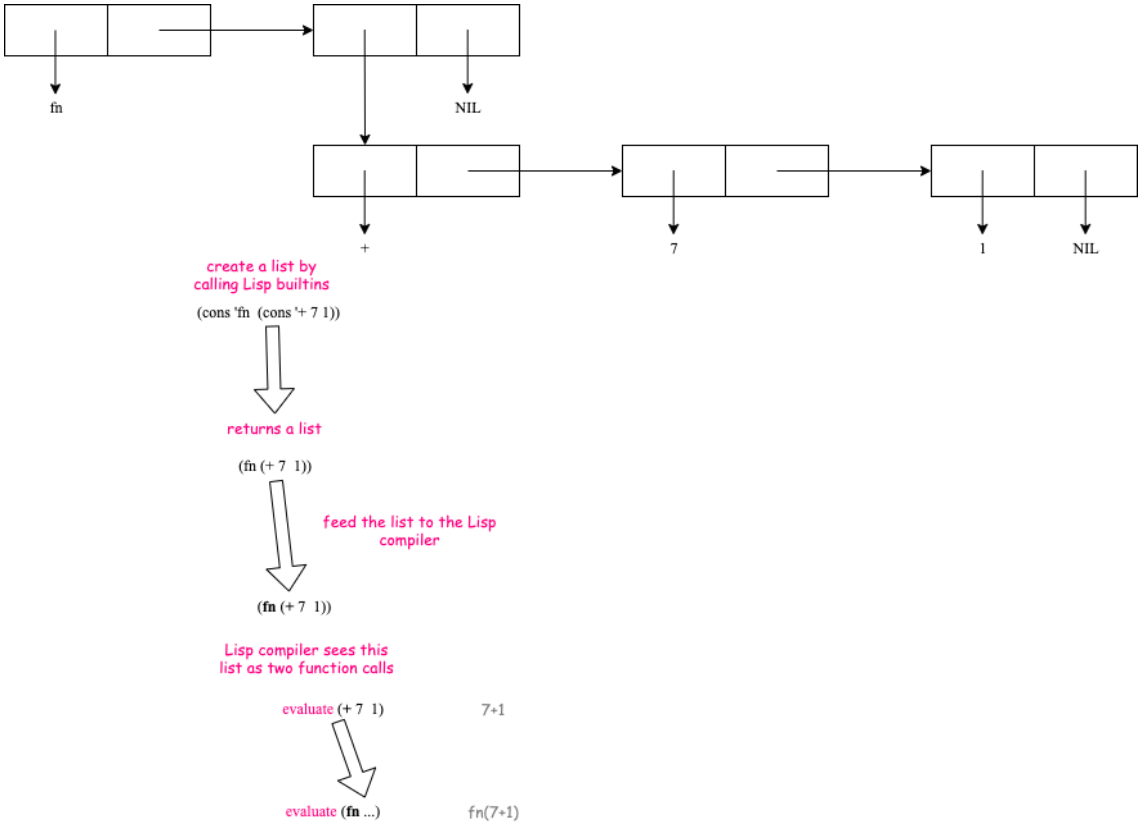
Other languages

fn((7+1))

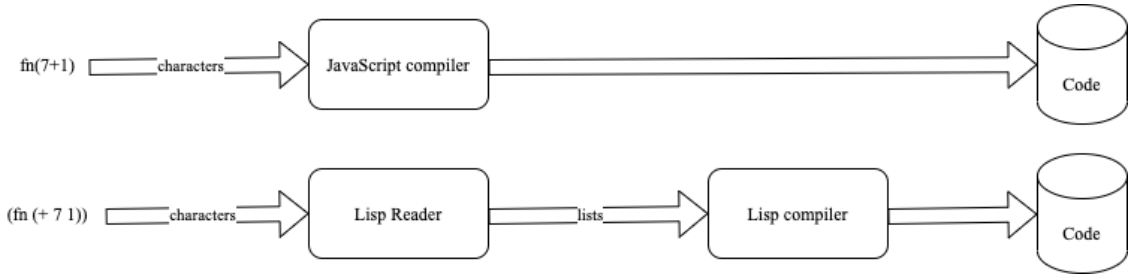
Written as

fn(7+1)

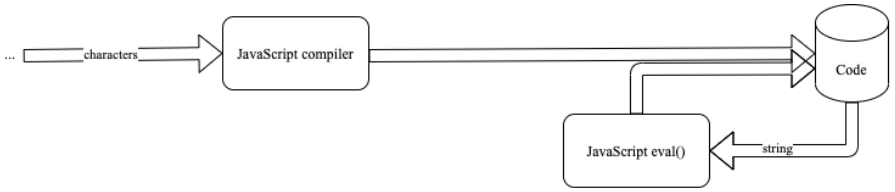
2. Creating Add



3. Compiler Basics



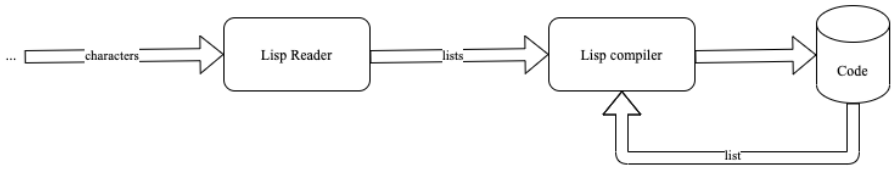
4. Programs That Write Programs



uncommon

frowned upon
"don't ever use eval()"

but, the JS compiler IS eval() ! *



very common

programs that write programs
are called "macros"

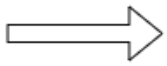
macros are very common - e.g.
keyboard macros, shortcuts,
abbreviations

lisp programmers create lists,
then feed the created lists to
the Lisp compiler

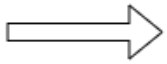
Lispers use macros so often that
they invented syntactic sugar to
help with creating list templates
- backquote, comma, comma-at

* You use eval() frequently, but
only to compile code that you've
written yourself.

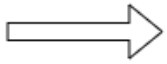
5. Macro Syntax (Syntactic Sugar)



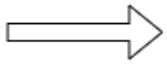
`(5) → (5) *



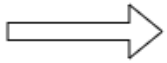
(setq x 7)
`(,x) → (7)
x = 7
make a list containing the value of x



(setq x '(1 2))
`(,x) → ((1 2))



(setq x '(1 2))
`(,@x) → (1 2)
list splice



(setq x '(1 2))
(setq y '(3 4))
`(,@x ,@y) → (1 2 3 4)

N.B single-quote ' was in original Lisp
back-quote ` was added later as
syntactic sugar

* back-quote ` is like single-quote ' if
there are no commas inside

(1 2 3 4) is a valid list, but, if you feed it to the Lisp compiler, the compiler will think that the first element, 1, is a function - but, 1 is a number, not a function, so the compiler will throw an error

5. Inc (Macro)

```
(defmacro my-inc (x)
  `(+ ,x 1))
```

syntactic sugar for

```
(defmacro my-inc (x)
  (cons '+ (cons x (cons 1 nil))))
```

"defmacro" means "compile-time function" - every time the compiler sees (my-inc ...) it will call the my-inc function and compile the returned list