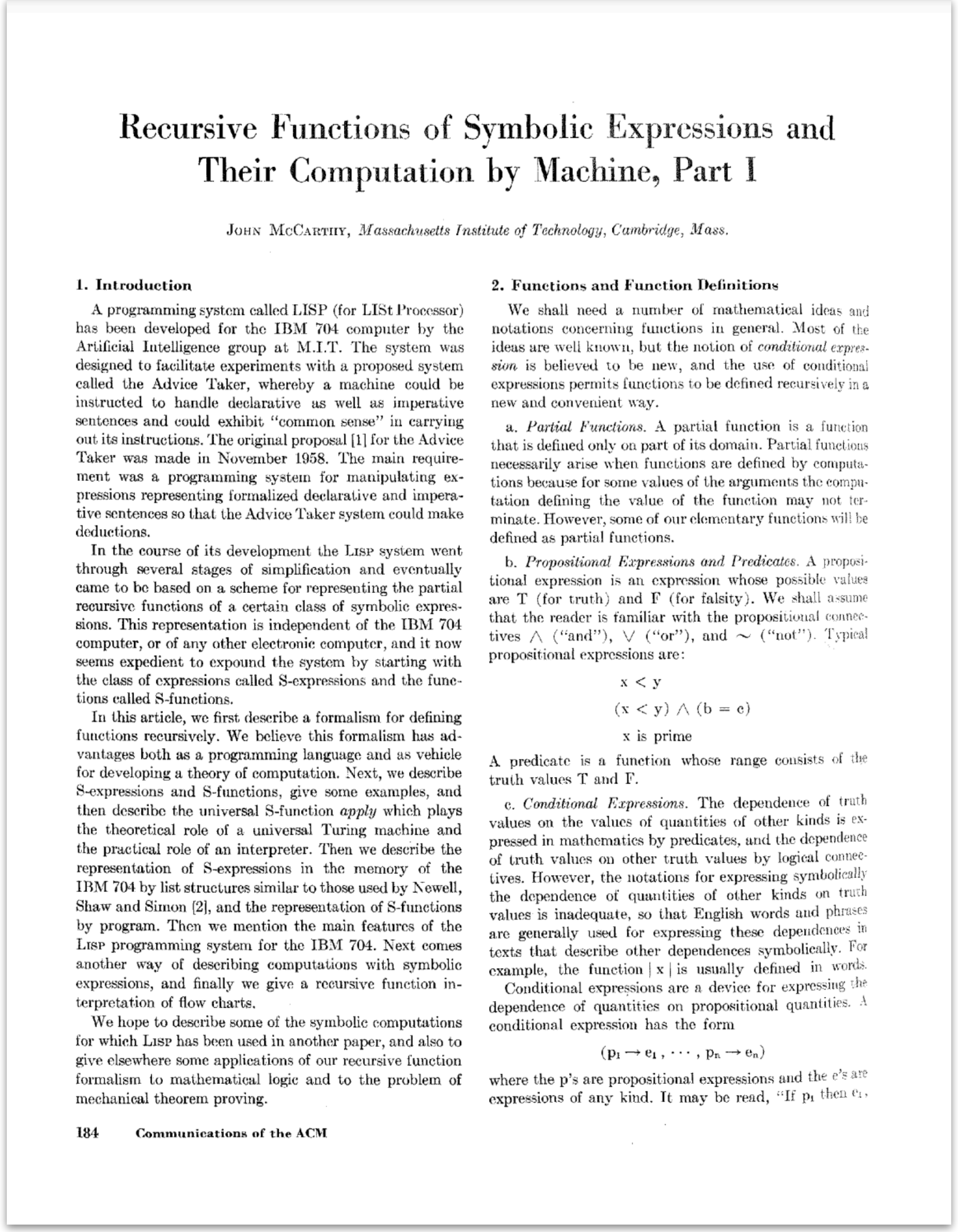
LISP from scratch on a PDP-11, Part 1

Jacques Comeaux

Why LISP?

- Influential
  - Recursion
  - Conditional expressions
  - Garbage collection
  - First-class functions
  - Symbols



Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

JOHN MCCARTHY, *Massachusetts Institute of Technology, Cambridge, Mass.*

1. Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

In this article, we first describe a formalism for defining functions recursively. We believe this formalism has advantages both as a programming language and as vehicle for developing a theory of computation. Next, we describe S-expressions and S-functions, give some examples, and then describe the universal S-function *apply* which plays the theoretical role of a universal Turing machine and the practical role of an interpreter. Then we describe the representation of S-expressions in the memory of the IBM 704 by list structures similar to those used by Newell, Shaw and Simon [2], and the representation of S-functions by program. Then we mention the main features of the LISP programming system for the IBM 704. Next comes another way of describing computations with symbolic expressions, and finally we give a recursive function interpretation of flow charts.

We hope to describe some of the symbolic computations for which LISP has been used in another paper, and also to give elsewhere some applications of our recursive function formalism to mathematical logic and to the problem of mechanical theorem proving.

2. Functions and Function Definitions

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known, but the notion of *conditional expression* is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

a. *Partial Functions.* A partial function is a function that is defined only on part of its domain. Partial functions necessarily arise when functions are defined by computations because for some values of the arguments the computation defining the value of the function may not terminate. However, some of our elementary functions will be defined as partial functions.

b. *Propositional Expressions and Predicates.* A propositional expression is an expression whose possible values are T (for truth) and F (for falsity). We shall assume that the reader is familiar with the propositional connectives $\wedge$ ("and"), $\vee$ ("or"), and $\sim$ ("not"). Typical propositional expressions are:

$$x < y$$

$$(x < y) \wedge (b = c)$$

$$x \text{ is prime}$$

A predicate is a function whose range consists of the truth values T and F.

c. *Conditional Expressions.* The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence of truth values on other truth values by logical connectives. However, the notations for expressing symbolically the dependence of quantities of other kinds on truth values is inadequate, so that English words and phrases are generally used for expressing these dependences in texts that describe other dependences symbolically. For example, the function $|x|$ is usually defined in words.

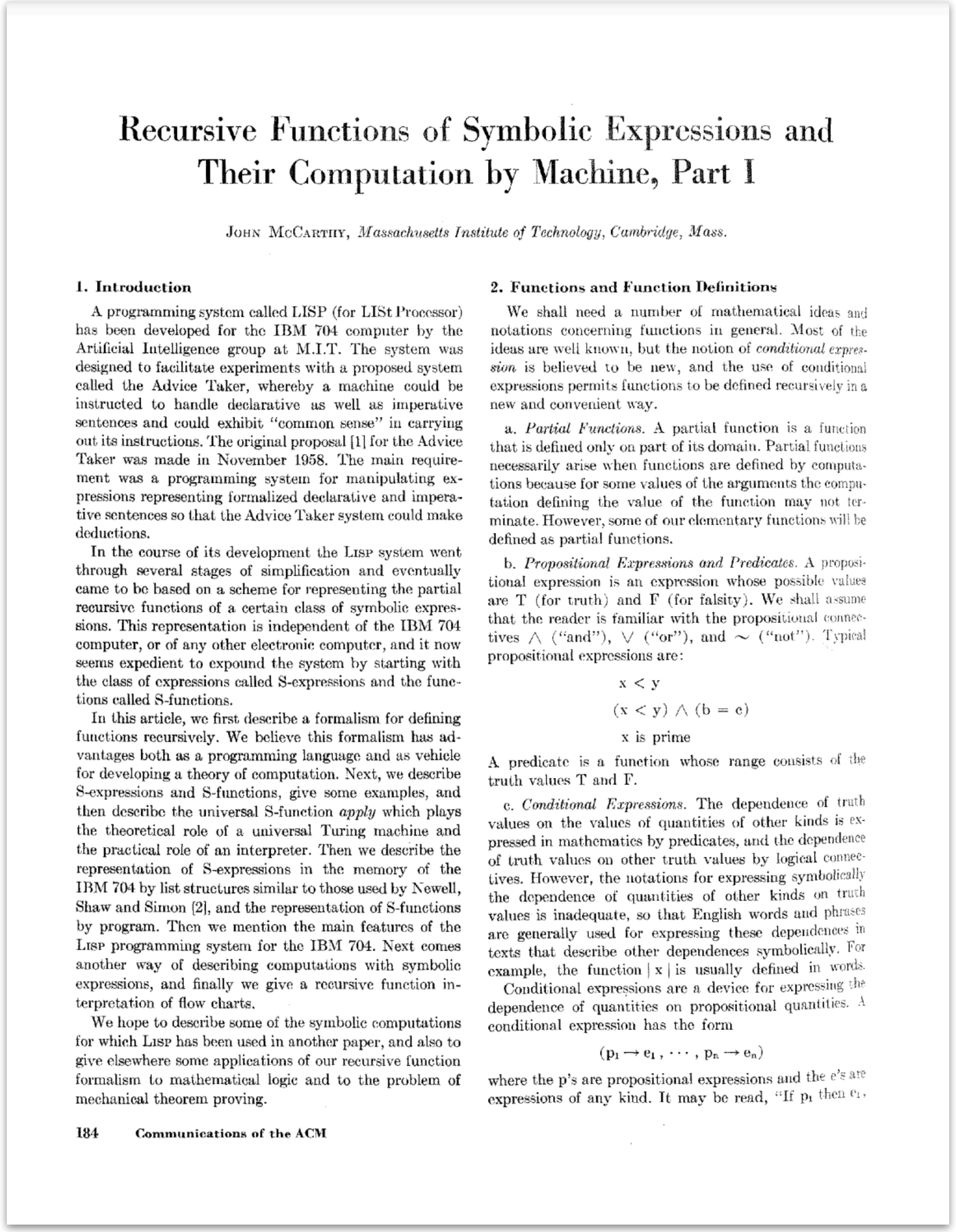
Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

where the p's are propositional expressions and the e's are expressions of any kind. It may be read, "If p_1 then e_1,

Why LISP?

- Influential
  - Recursion
  - Conditional expressions
  - Garbage collection
  - First-class functions
  - Symbols
- **Simple**



Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

JOHN MCCARTHY, *Massachusetts Institute of Technology, Cambridge, Mass.*

1. Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

In this article, we first describe a formalism for defining functions recursively. We believe this formalism has advantages both as a programming language and as vehicle for developing a theory of computation. Next, we describe S-expressions and S-functions, give some examples, and then describe the universal S-function *apply* which plays the theoretical role of a universal Turing machine and the practical role of an interpreter. Then we describe the representation of S-expressions in the memory of the IBM 704 by list structures similar to those used by Newell, Shaw and Simon [2], and the representation of S-functions by program. Then we mention the main features of the LISP programming system for the IBM 704. Next comes another way of describing computations with symbolic expressions, and finally we give a recursive function interpretation of flow charts.

We hope to describe some of the symbolic computations for which LISP has been used in another paper, and also to give elsewhere some applications of our recursive function formalism to mathematical logic and to the problem of mechanical theorem proving.

2. Functions and Function Definitions

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known, but the notion of *conditional expression* is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

a. *Partial Functions.* A partial function is a function that is defined only on part of its domain. Partial functions necessarily arise when functions are defined by computations because for some values of the arguments the computation defining the value of the function may not terminate. However, some of our elementary functions will be defined as partial functions.

b. *Propositional Expressions and Predicates.* A propositional expression is an expression whose possible values are T (for truth) and F (for falsity). We shall assume that the reader is familiar with the propositional connectives $\wedge$ ("and"), $\vee$ ("or"), and $\sim$ ("not"). Typical propositional expressions are:

$$x < y$$

$$(x < y) \wedge (b = c)$$

$$x \text{ is prime}$$

A predicate is a function whose range consists of the truth values T and F.

c. *Conditional Expressions.* The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence of truth values on other truth values by logical connectives. However, the notations for expressing symbolically the dependence of quantities of other kinds on truth values is inadequate, so that English words and phrases are generally used for expressing these dependences in texts that describe other dependences symbolically. For example, the function $|x|$ is usually defined in words.

Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

where the p's are propositional expressions and the e's are expressions of any kind. It may be read, "If p_1 then e_1,

184 Communications of the ACM

S-Expressions

- Symbolic expressions
- Each S-Expression is either
  - An atomic symbol
  - A list of S-Expressions

S-Expressions

- Symbolic expressions
- Each S-Expression is either
  - An atomic symbol
  - A list of S-Expressions

```
DOG
```

S-Expressions

- Symbolic expressions
- Each S-Expression is either
  - An atomic symbol
  - A list of S-Expressions

APPLE

DOG

S-Expressions

- Symbolic expressions
- Each S-Expression is either
  - An atomic symbol
  - A list of S-Expressions

APPLE

DOG

LISP

S-Expressions

- Symbolic expressions
- Each S-Expression is either
  - An atomic symbol
  - A list of S-Expressions

APPLE

DOG

LISP

(A B C D E)

S-Expressions

- Symbolic expressions
- Each S-Expression is either
  - An atomic symbol
  - A list of S-Expressions

APPLE

DOG

LISP

(A B C D E)

((X Y) (Y Z) (P Q R))

S-Expressions

- Symbolic expressions
- Each S-Expression is either
  - An atomic symbol
  - A list of S-Expressions

APPLE

DOG

LISP

(A B C D E)

((X Y) (Y Z) (P Q R))

(THIS (IS AN) S EXPRESSION)

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
  - atom

atom [X] = T

atom [(X …)] = F

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
  - atom
  - eq

atom [X] = T

atom [(X …)] = F

eq [x; y] = T if x and y are the same symbol

eq [x; y] = F otherwise

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
  - atom
  - eq
  - car

atom [X] = T

atom [(X …)] = F

eq [x; y] = T if x and y are the same symbol

eq [x; y] = F otherwise

car [(X_1 X_2 … X_n)] = X_1

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
  - atom
  - eq
  - car
  - cdr

atom [X] = T

atom [(X …)] = F

eq [x; y] = T if x and y are the same symbol

eq [x; y] = F otherwise

car [(X_1 X_2 … X_n)] = X_1

cdr [(X_1 X_2 … X_n)] = (X_2 … X_n)

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
  - atom
  - eq
  - car
  - cdr
  - cons

atom [X] = T

atom [(X …)] = F

eq [x; y] = T if x and y are the same symbol

eq [x; y] = F otherwise

car [(X_1 X_2 … X_n)] = X_1

cdr [(X_1 X_2 … X_n)] = (X_2 … X_n)

cons [X_1; (X_2 … X_n)] = (X_1 X_2 … X_n)

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
- Conditional expressions
  - Evaluate p's from left to right
  - Result is the e corresponding to the first p which evaluates to T

$$[p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n]$$

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
- Conditional expressions
  - Evaluate p's from left to right
  - Result is the e corresponding to the first p which evaluates to T

atom

eq

car

cdr

cons

$[p_1 \to e_1, \cdots, p_n \to e_n]$

S-Functions

- Functions of symbolic expressions
- Five primitive S-Functions
- Conditional expressions
  - Evaluate p's from left to right
  - Result is the e corresponding to the first p which evaluates to T
- Arbitrary S-Functions?

atom

eq

car

cdr

cons

$[p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n]$

?

Interlude:

Lambda Calculus!

Lambda Calculus

- A formal system for functions
- Three syntactic constructs

Lambda Calculus

- A formal system for functions
- Three syntactic constructs
  - Variables

$$x$$

Lambda Calculus

- A formal system for functions
- Three syntactic constructs
  - Variables
  - Functions

$$\lambda x \,.\, N$$

Lambda Calculus

- A formal system for functions
- Three syntactic constructs
  - Variables
  - Functions
  - Application

$$MN$$

Lambda Calculus

- A formal system for functions
- Three syntactic constructs
  - Variables
  - Functions
  - Application

$$x$$

$$\lambda x \,.\, N$$

$$MN$$

Lambda Calculus

- A formal system for functions
- Three syntactic constructs
  - Variables
  - Functions
  - Application
- Sometimes more
  - Booleans; Natural numbers; etc.

$$x$$

$$\lambda x \, . \, N$$

$$MN$$

Lambda Calculus

- A formal system for functions
- Three syntactic constructs
  - Variables
  - Functions
  - Application
- Sometimes more
  - Booleans; Natural numbers; etc.

$$x$$

$$\lambda x\,.\,N$$

$$MN$$

$$0, 1, 2, \ldots$$

Lambda Calculus

- A formal system for functions
- Three syntactic constructs
  - Variables
  - Functions
  - Application
- Sometimes more
  - Booleans; Natural numbers; etc.

$$x$$

$$0, 1, 2, \ldots$$

$$\lambda x \, . \, N$$

$$M + N$$

$$MN$$

Lambda can’t hurt you

λ

“Normal” math

$$f(x) = x^2 + 5$$

“Normal” math

$$x \to x^2 + 5$$

Lambda notation

$$\lambda x \,.\, x^2 + 5$$

Substitution example

$$(\lambda x \,.\, x^2 + 5)\; N$$

$$N^2 + 5$$

Example Lambda calculus expressions

$$\lambda f . \lambda x . \lambda y . fyx$$

$$\lambda f . \lambda g . \lambda x . f(gx)$$

$$\lambda x . x$$

$$\lambda x . xx$$

$$(\lambda x . xx)(\lambda a . \lambda b . a)$$

Example Lambda calculus expressions

$$\lambda fxy\,.\,fyx$$

$$\lambda fgx\,.\,f(gx)$$

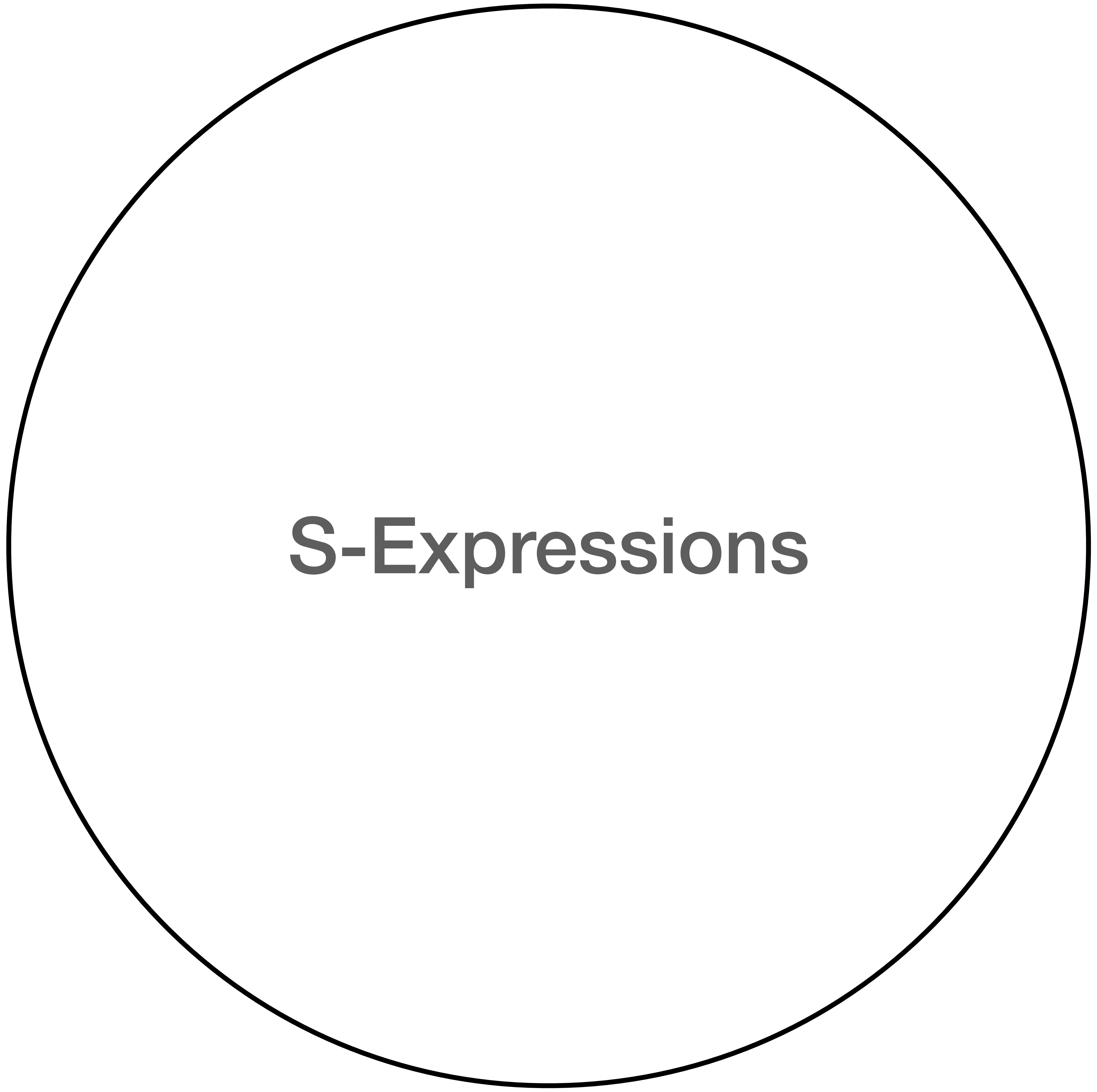
$$\lambda x\,.\,x$$

$$\lambda x\,.\,xx$$

$$(\lambda x\,.\,xx)(\lambda ab\,.\,a)$$

Back to LISP

S-Functions manipulate S-Expressions

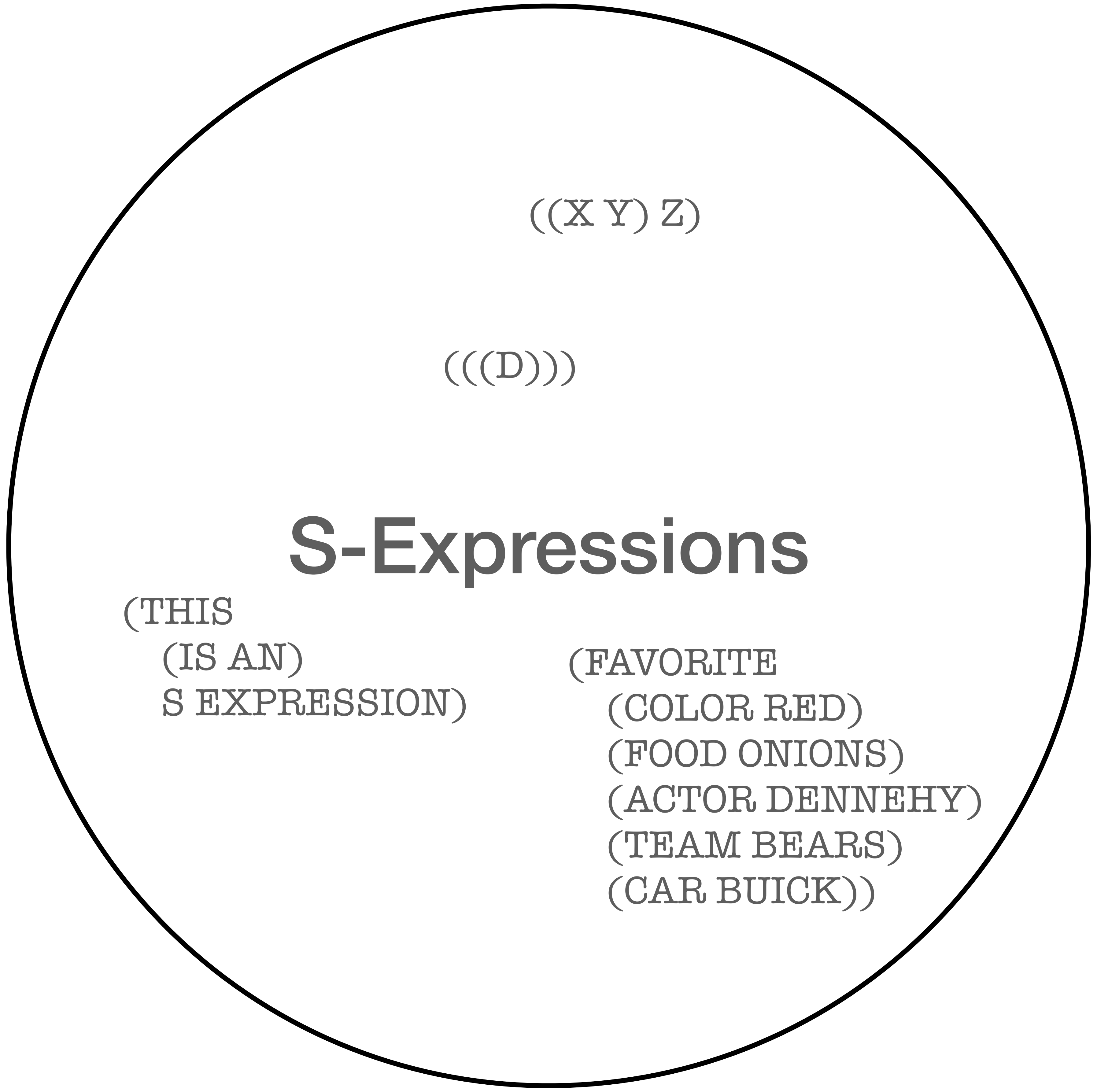


S-Functions manipulate S-Expressions

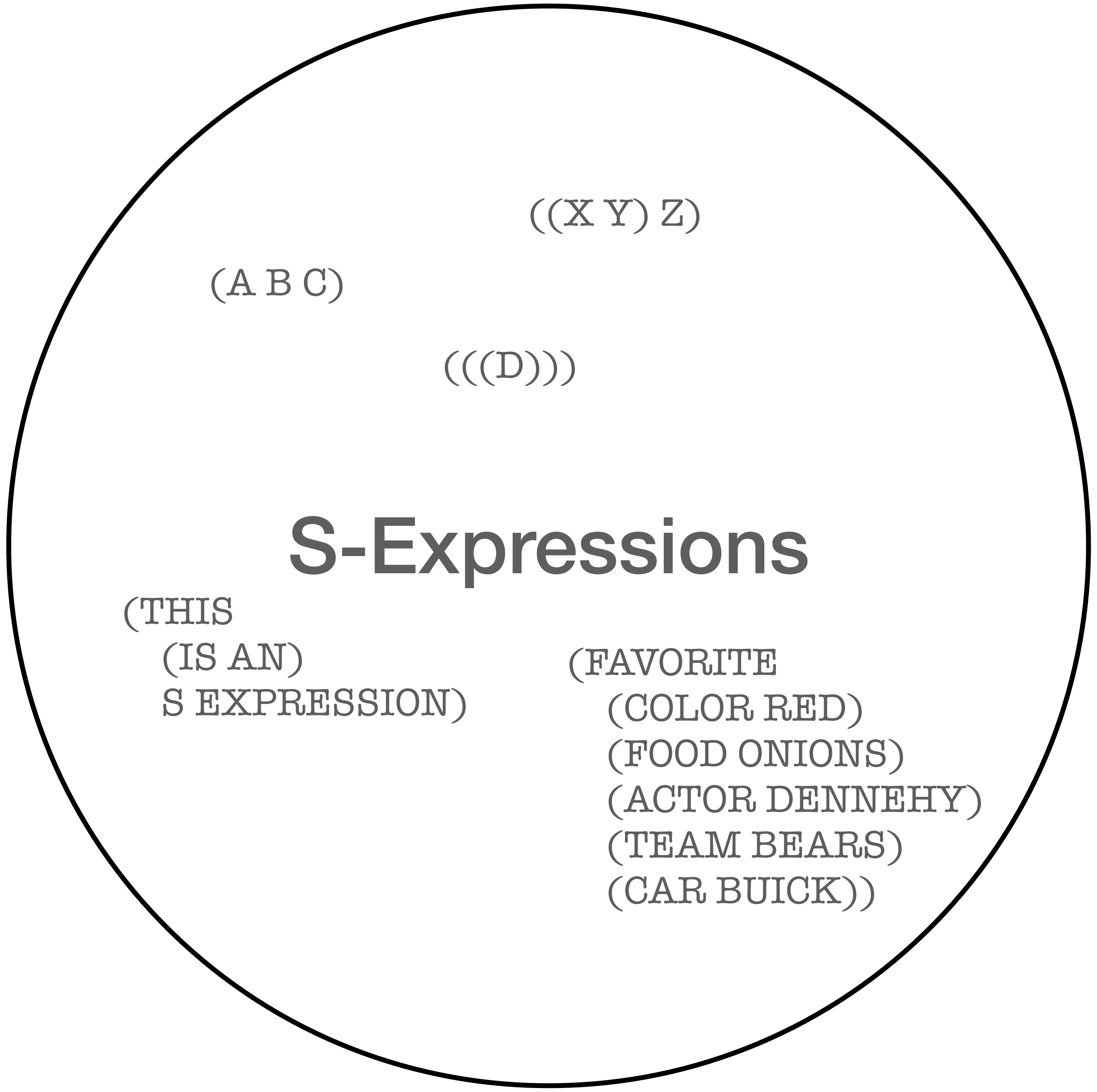
S-Functions manipulate S-Expressions

S-Functions manipulate S-Expressions

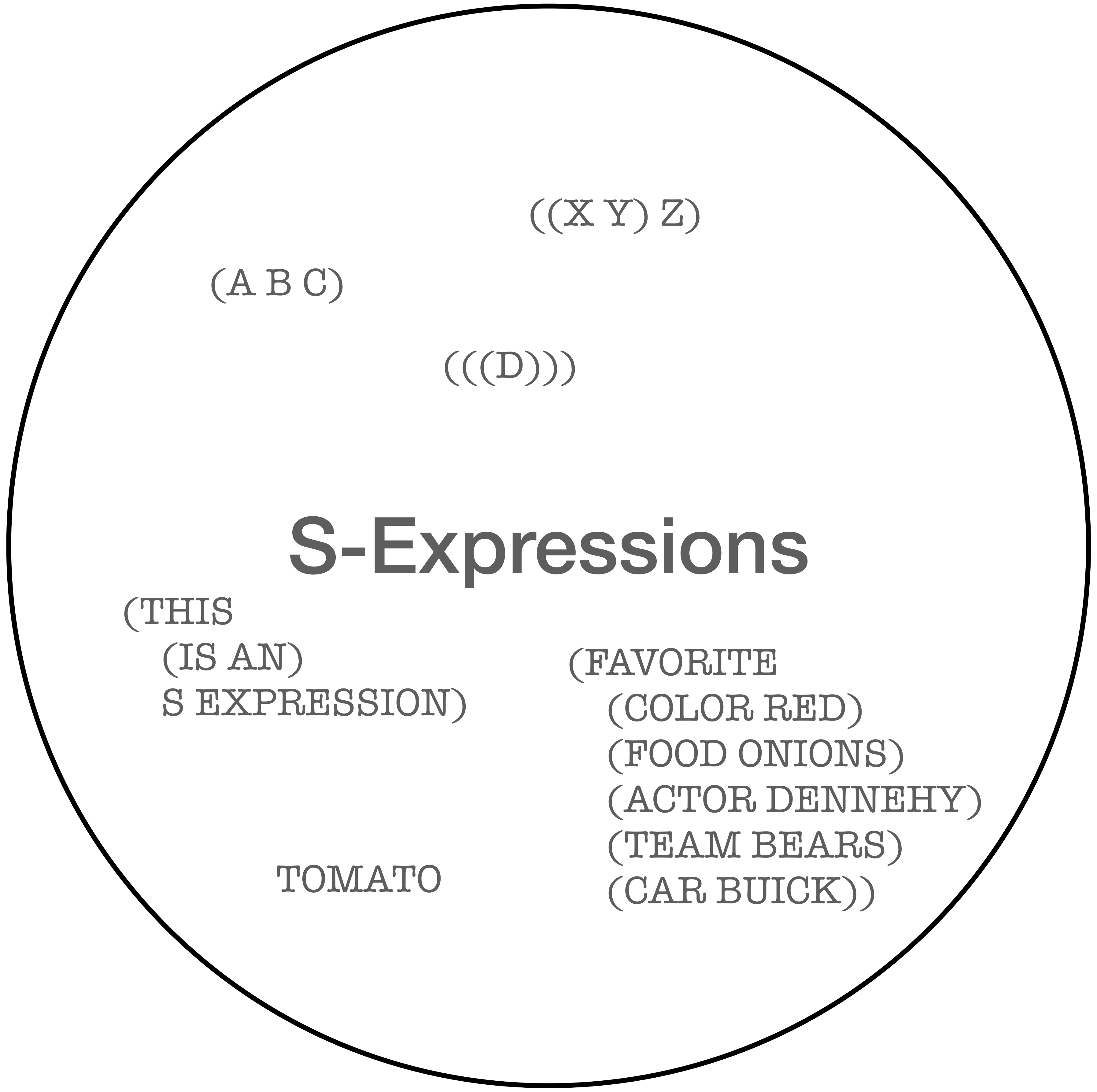
S-Functions manipulate S-Expressions



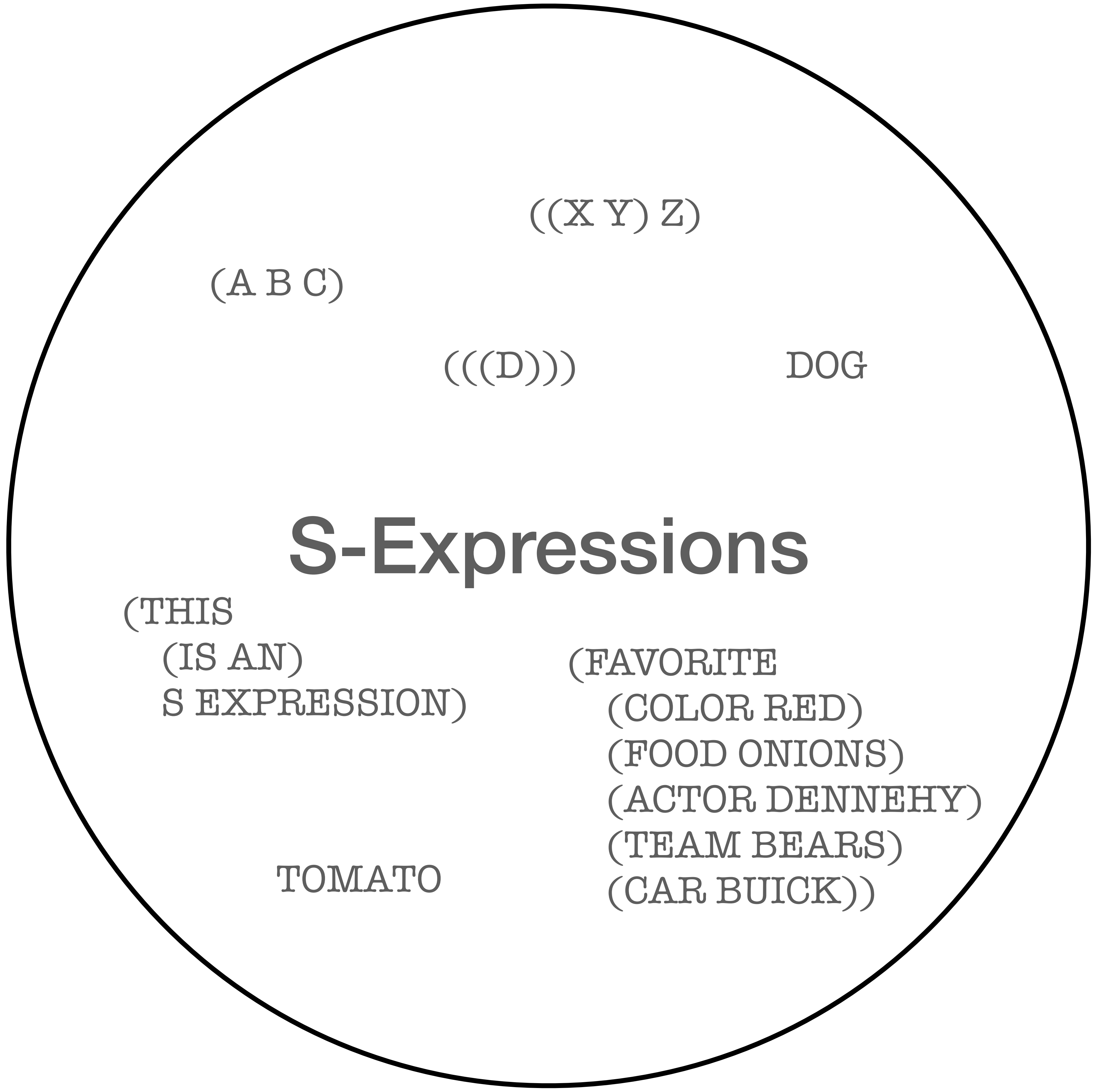
S-Functions manipulate S-Expressions



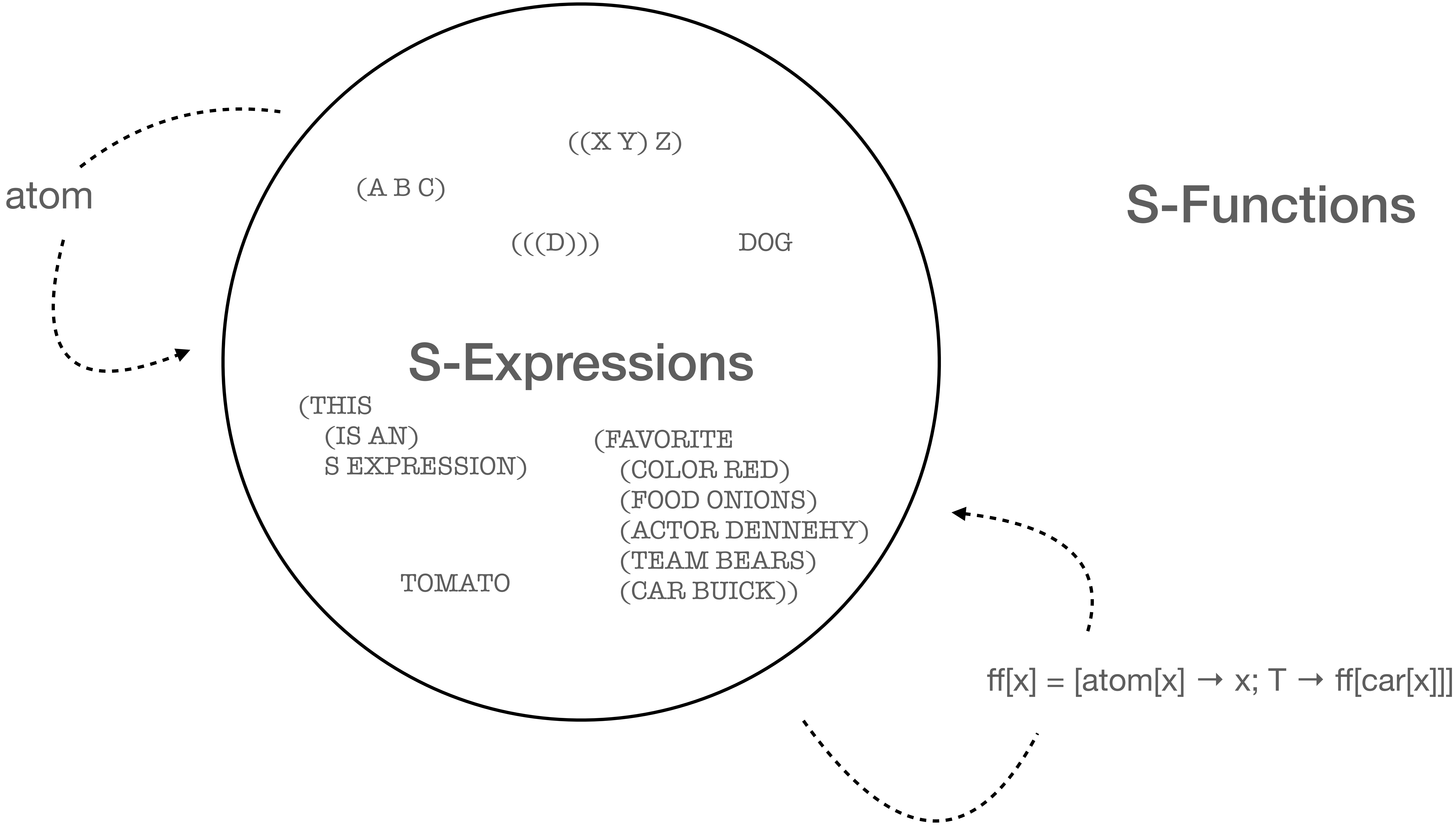
S-Functions manipulate S-Expressions



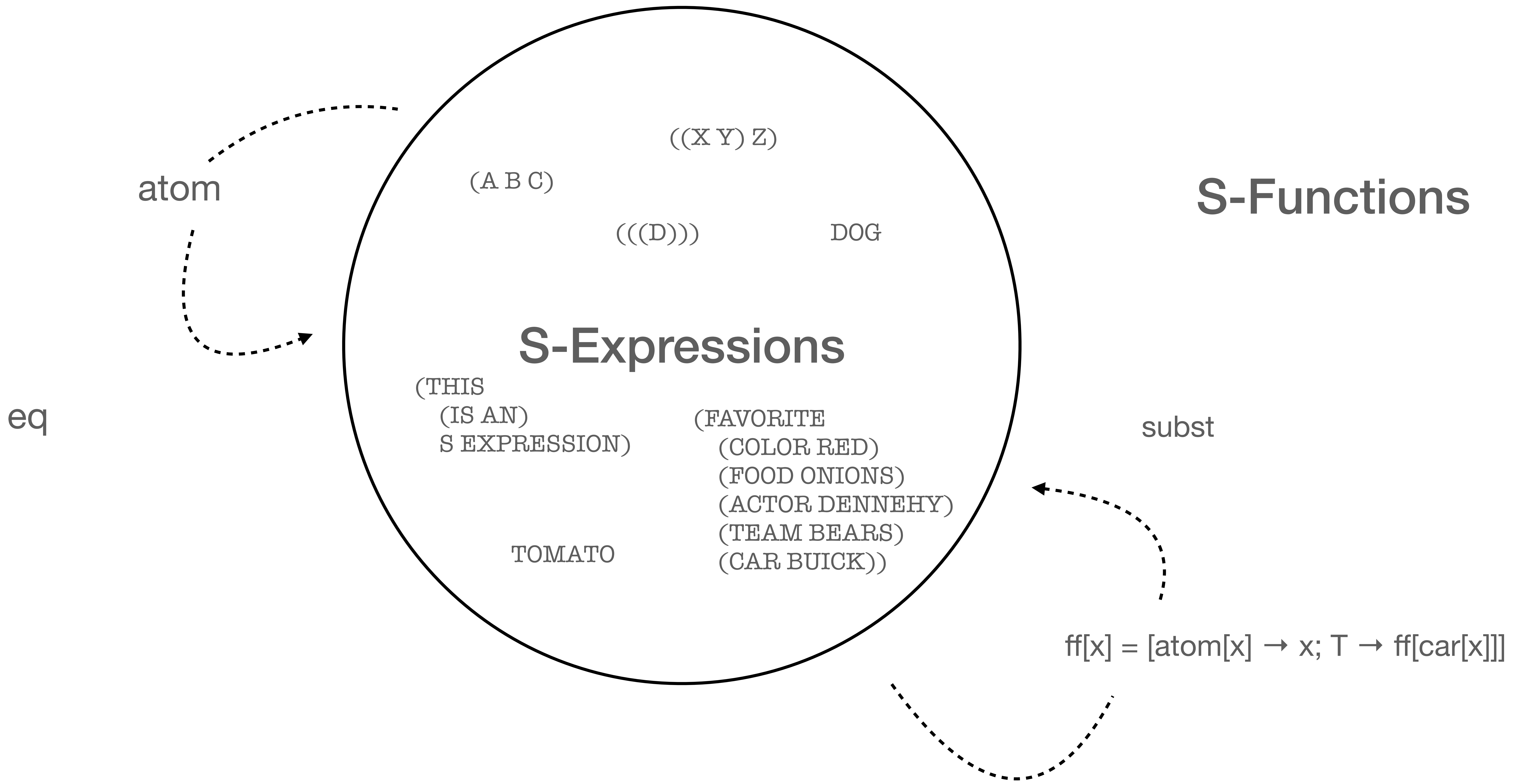
S-Functions manipulate S-Expressions



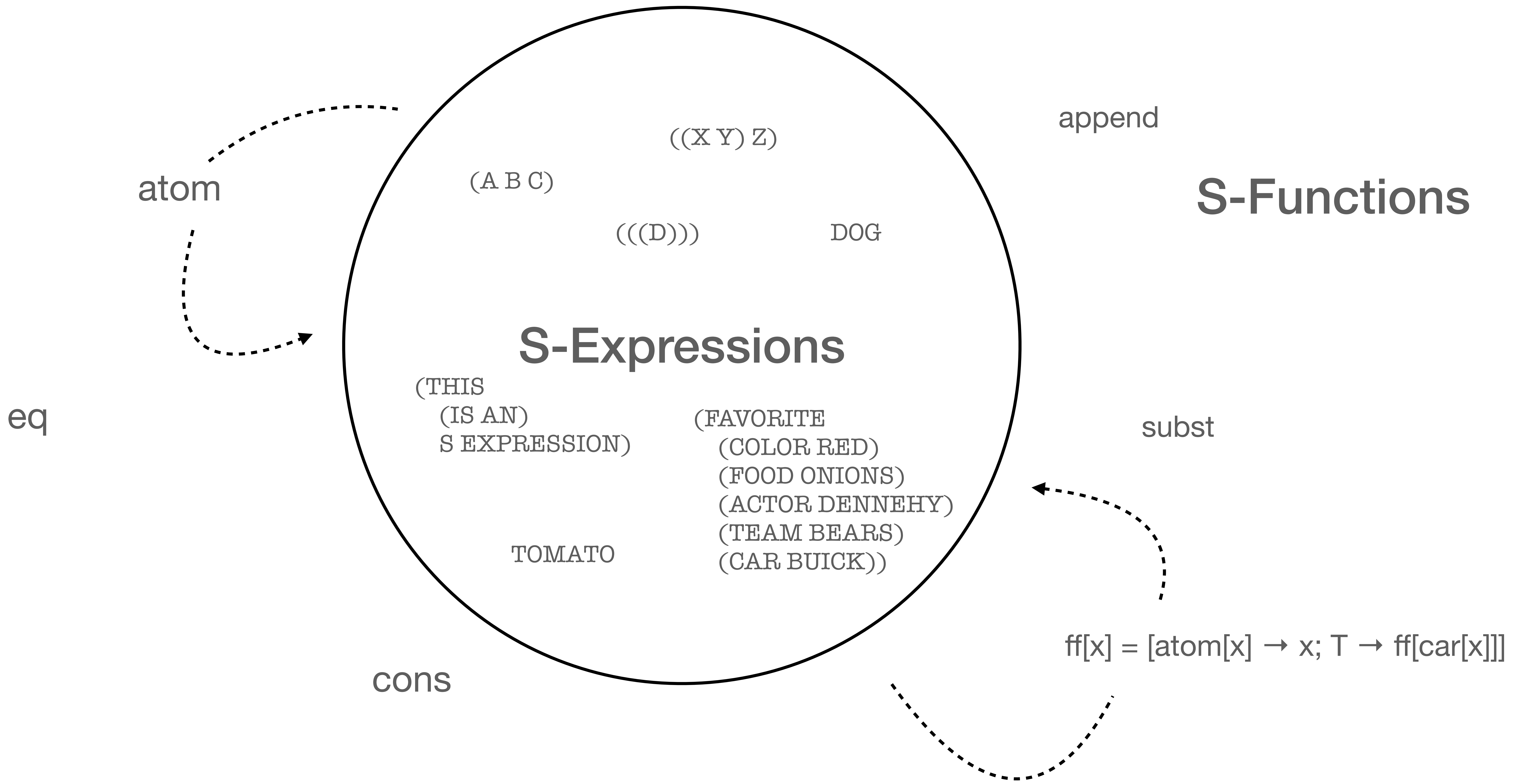
S-Functions manipulate S-Expressions



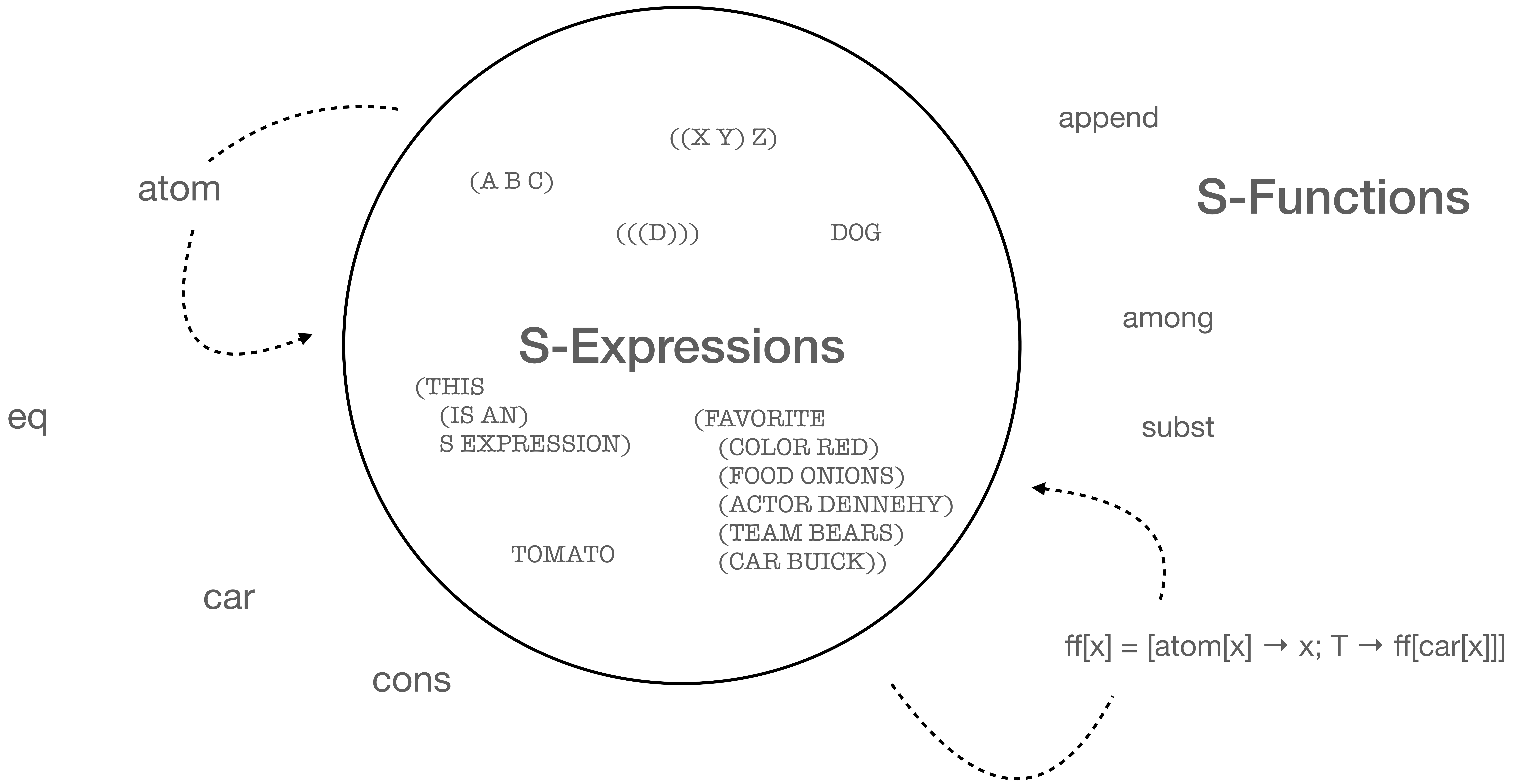
S-Functions manipulate S-Expressions



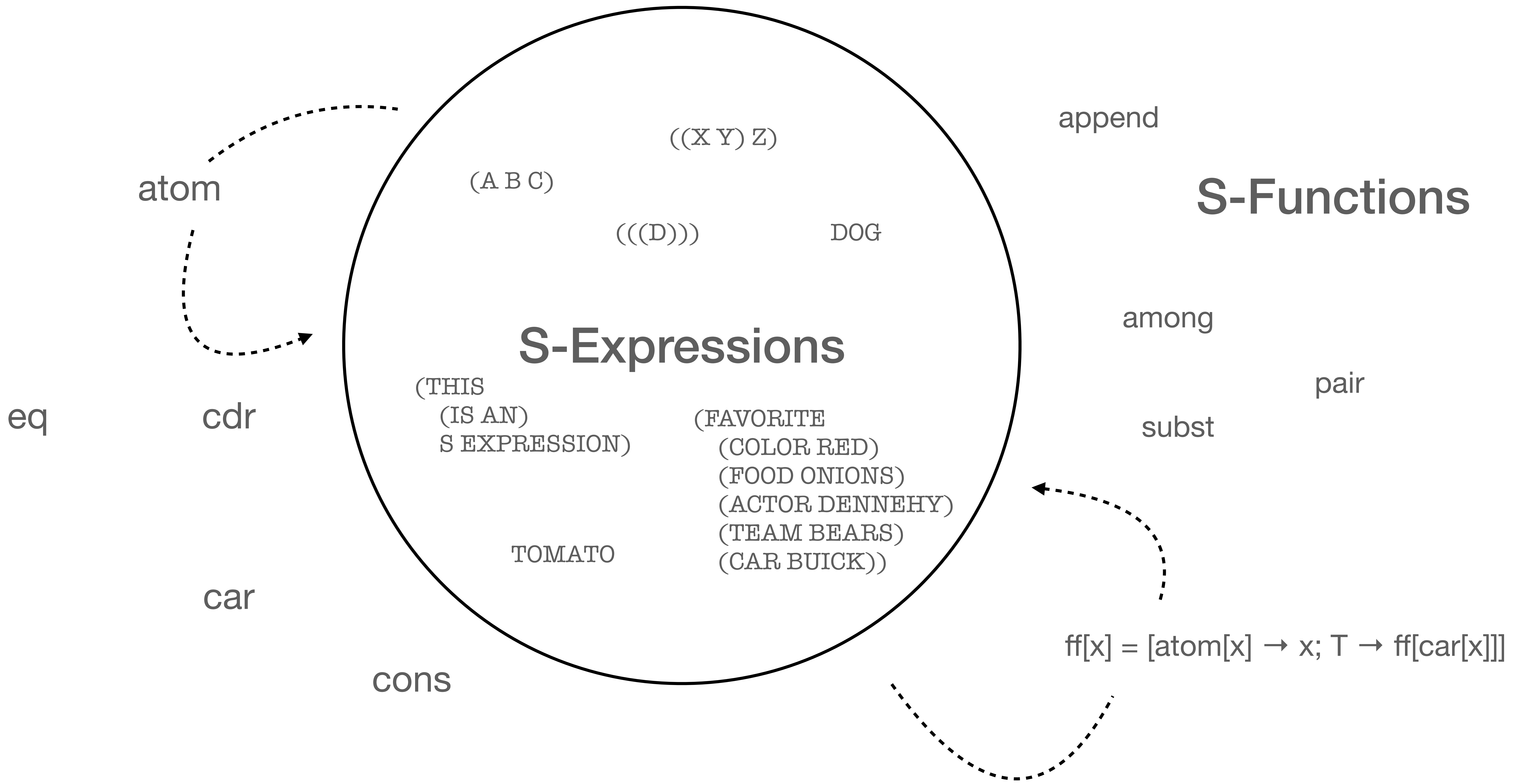
S-Functions manipulate S-Expressions



S-Functions manipulate S-Expressions



S-Functions manipulate S-Expressions



S-Functions *are* S-Expressions

subst [x; y; z] =

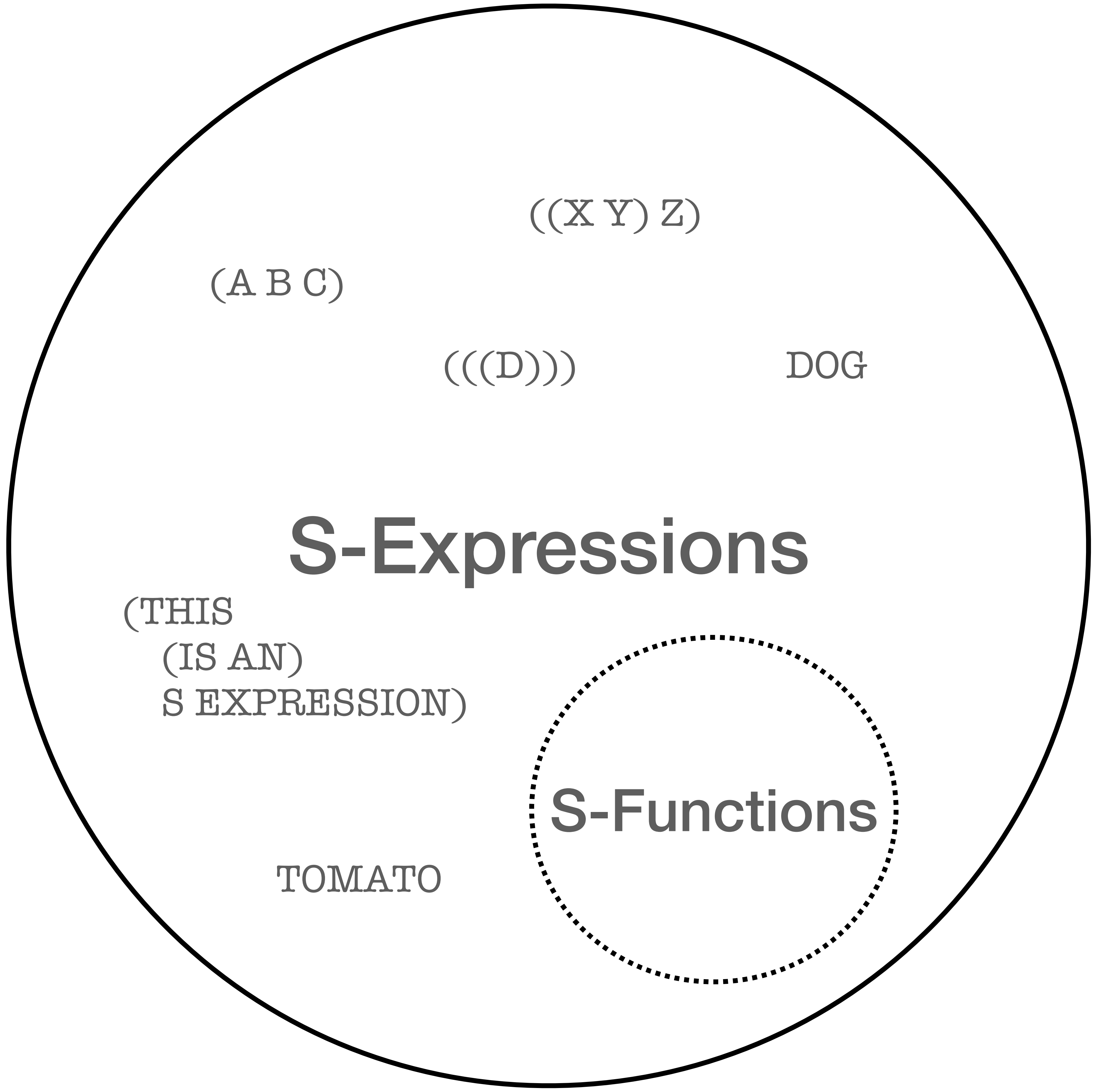
[atom (z) → [eq (y; z) → x; T → z]

; T → cons [subst [x; y; car [z]]; subst [x; y; cdr [z]]]]

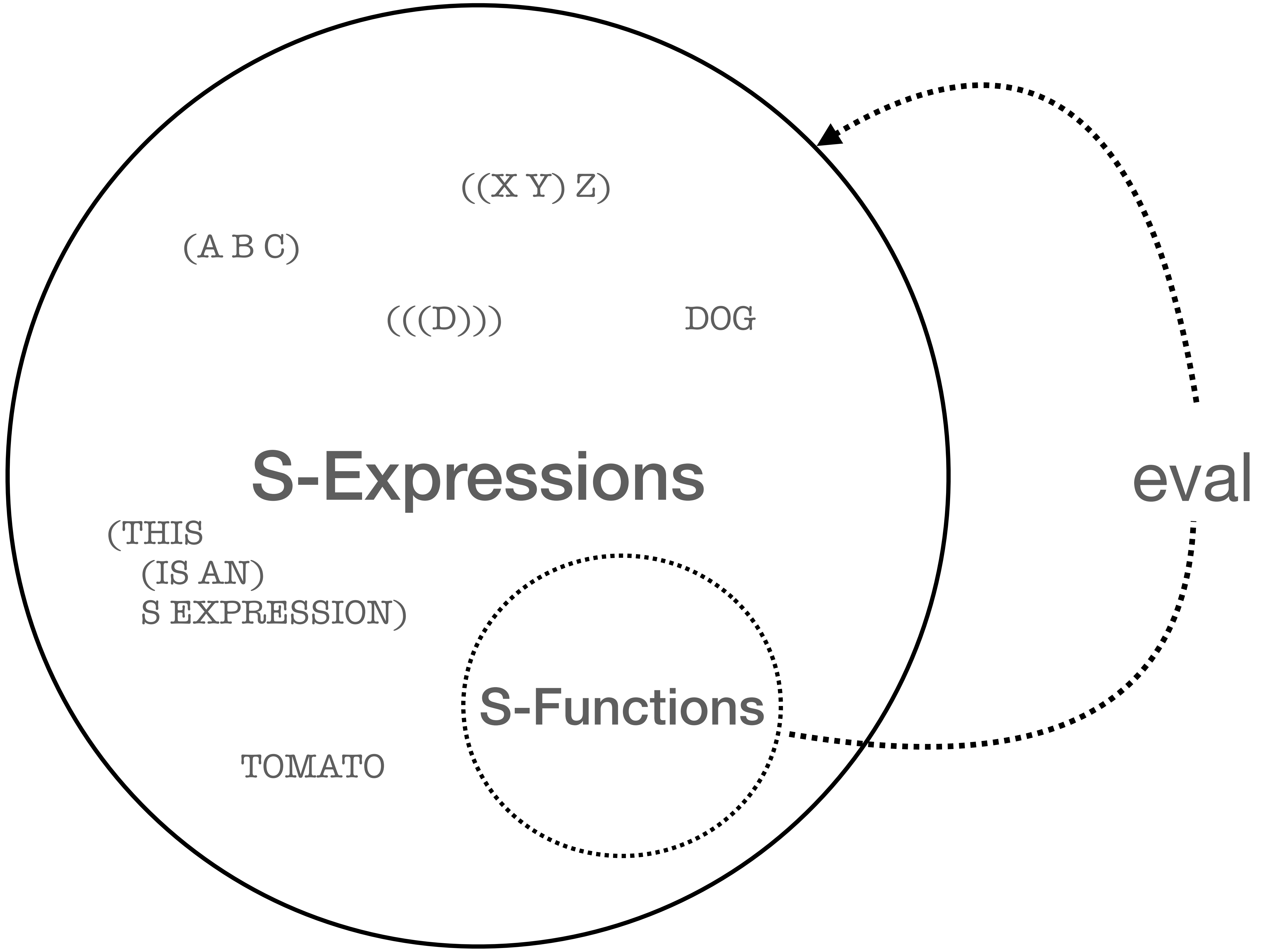
↓

(LABEL, SUBST, (LAMBDA, (X, Y, Z), (COND ((ATOM, Z), (COND, (EQ, Y, Z), X), ((QUOTE, T), Z))), ((QUOTE, T), (CONS, (SUBST, X, Y, (CAR Z)), (SUBST, X, Y, (CDR, Z)))))))

S-Functions *are* S-Expressions



What if? A universal evaluator

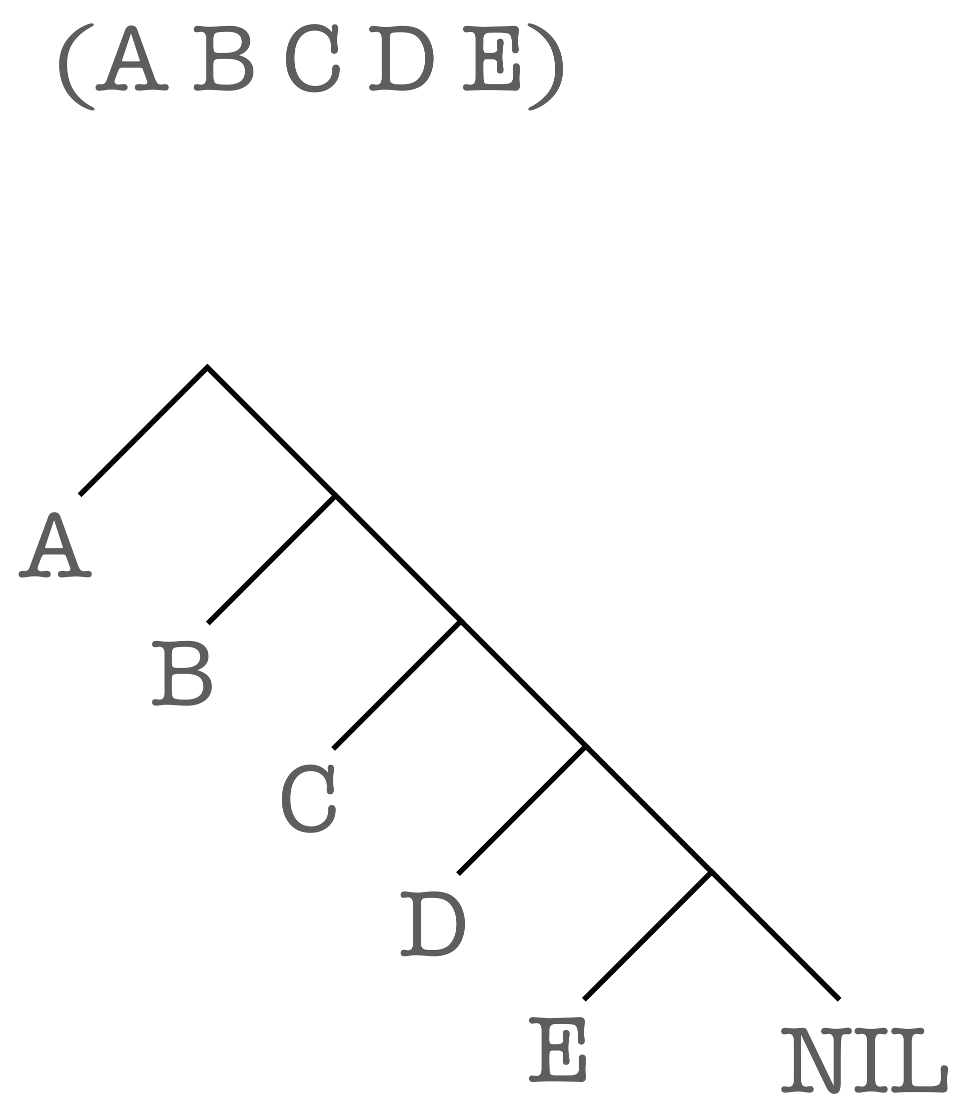


The universal evaluator

```
eval[e; a] =
    [ atom[e] → assoc[e; a]
    ; atom[car[e]] →
          [ eq[car[e]; QUOTE] → cadr[e]
          ; eq[car[e]; ATOM]  → atom[eval[cadr[e]; a]]
          ; eq[car[e]; EQ]    → eq[eval[cadr[e]; a]; eval[caddr[e]; a]]
          ; eq[car[e]; COND]  → evcon[cadr[e]; a]]
          ; eq[car[e]; CAR]   → car[eval[cadr[e]; a]]
          ; eq[car[e]; CDR]   → cdr[eval[cadr[e]; a]]
          ; eq[car[e]; CONS]  → cons[eval[cadr[e]; a]; eval[caddr[e]; a]]
          ; T                 → eval[cons[assoc[car[e]; a]; cdr[e]]; a]
          ]
      ]
    ; eq[caar[e]; LABEL]  → eval[cons[caddar[e]; cdr[e]]; cons[list[cadar[e]; car[e]]; a]]
    ; eq[caar[e]; LAMBDA] → eval[caddar[e]; append[pair[cadar[e]; evlis[cdr[e]; a]]; a]]
    ]
  where
    evcon[c; a] = [eval[caar[c]; a] → eval[cadar[c]; a]; T → evcon[cdr[c];a]]
    evlis[m; a] = [null[m] → NIL; T → cons[eval[car[m]; a]; evlis[cdr[m]; a]]]
```

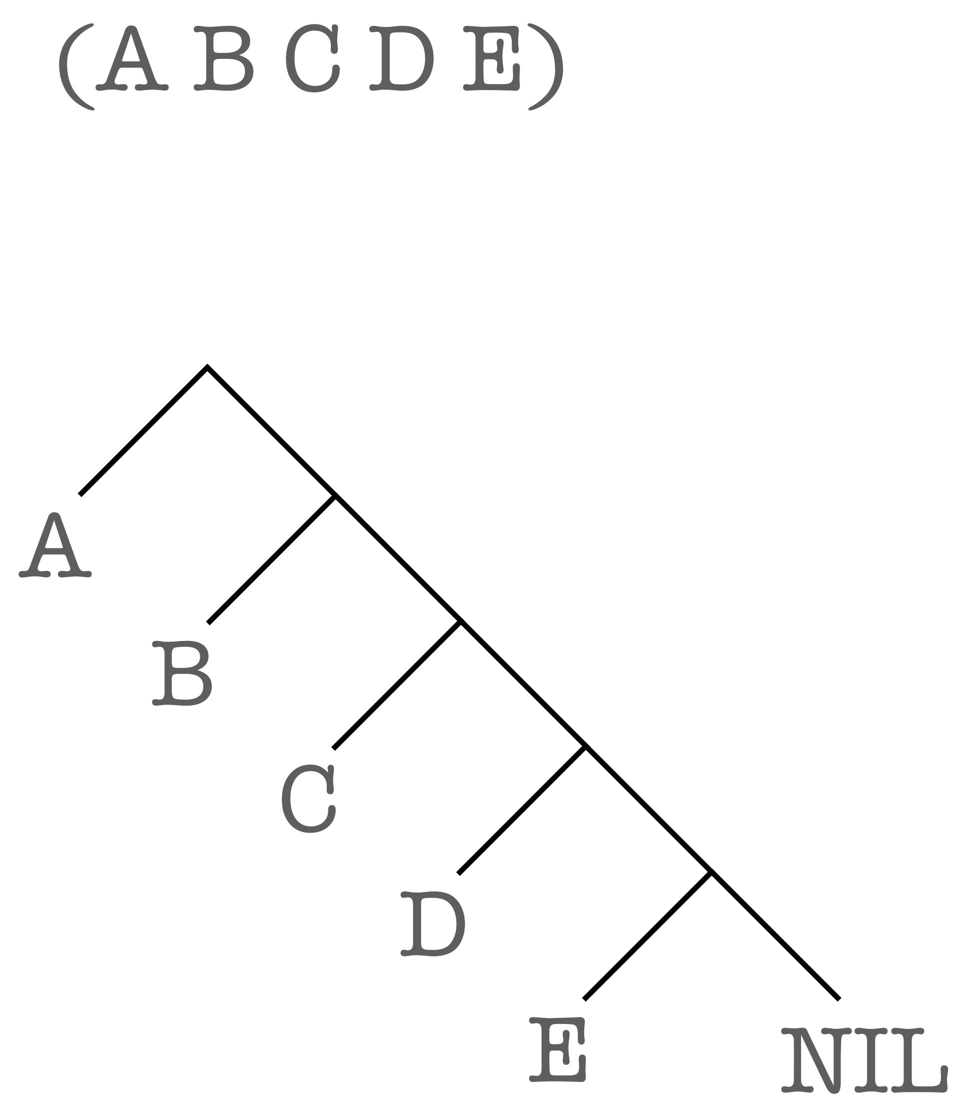
S-Expressions in memory

- Represent S-Expressions as binary trees
  - Left child is CAR (head)
  - Right child is CDR (tail)
- Leaf nodes are atoms



S-Expressions in memory

- Represent S-Expressions as binary trees
  - Left child is CAR (head)
  - Right child is CDR (tail)
- Leaf nodes are atoms
- LSB of pointer used to distinguish interior nodes from atoms

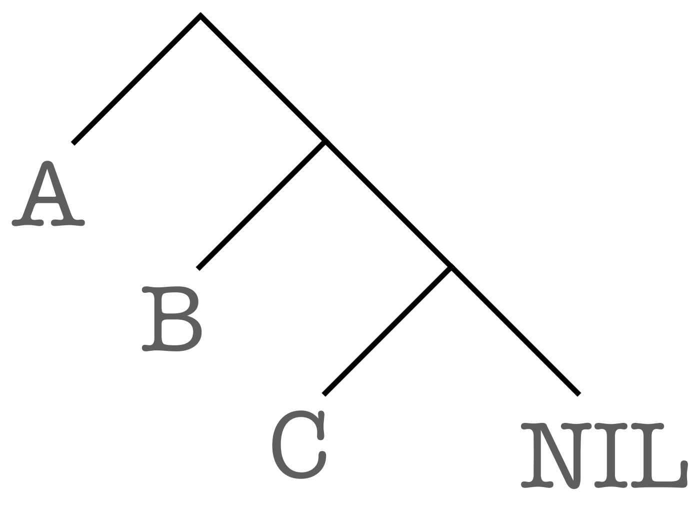


S-Expressions in memory

```
(A B C)
```

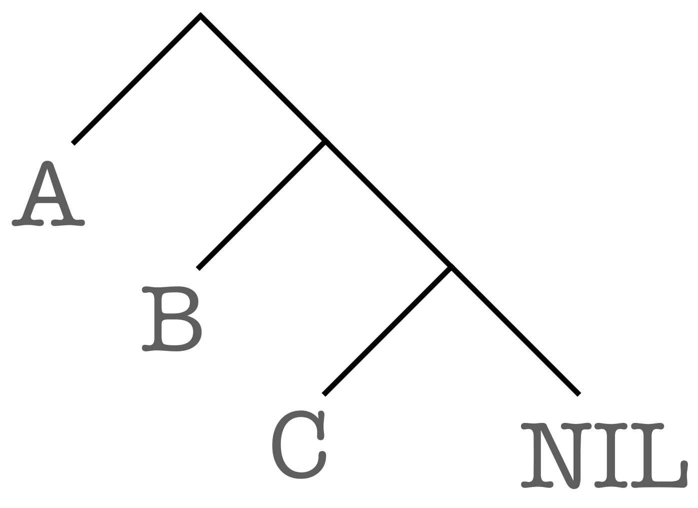
S-Expressions in memory

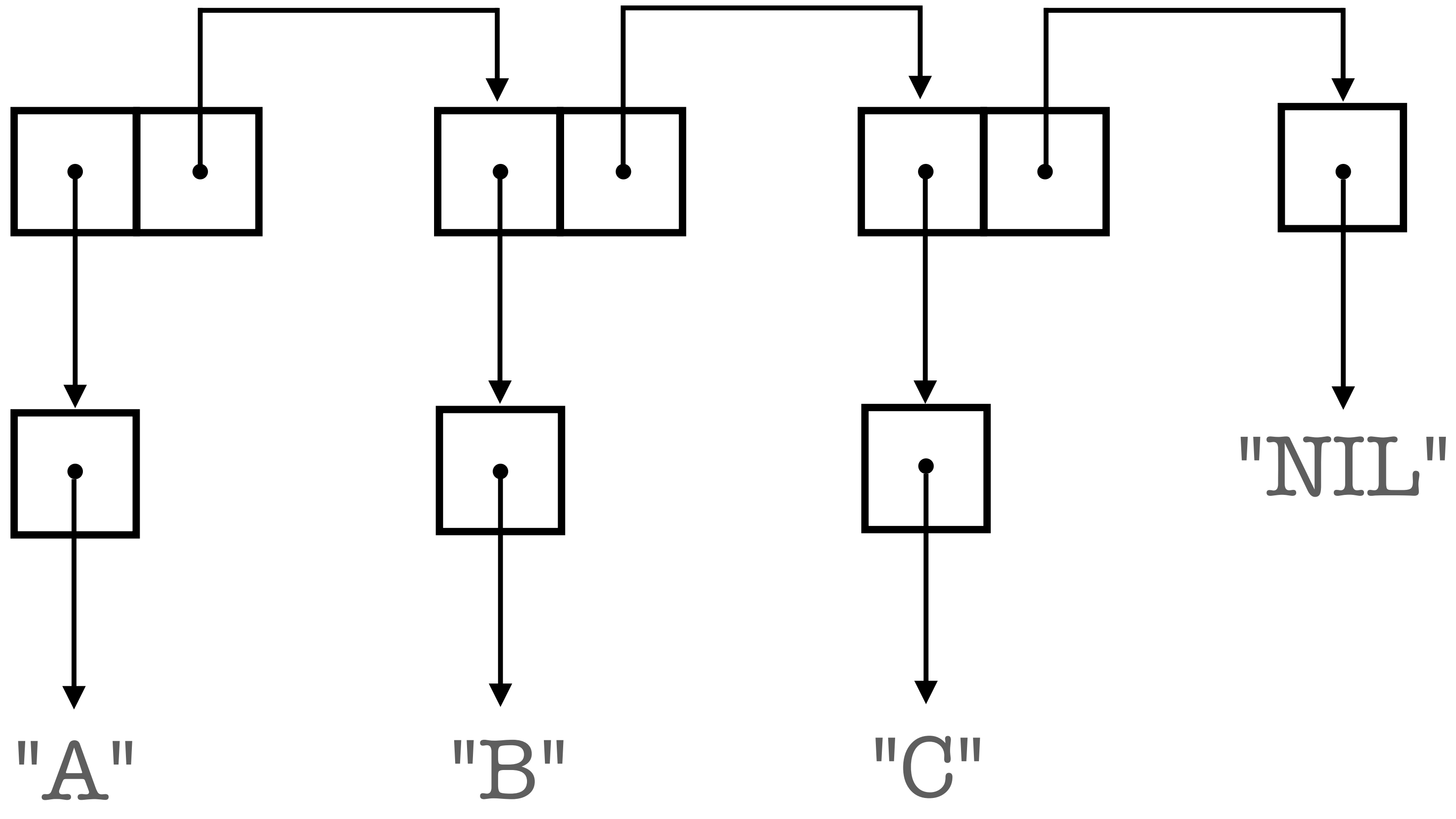
(A B C)



S-Expressions in memory

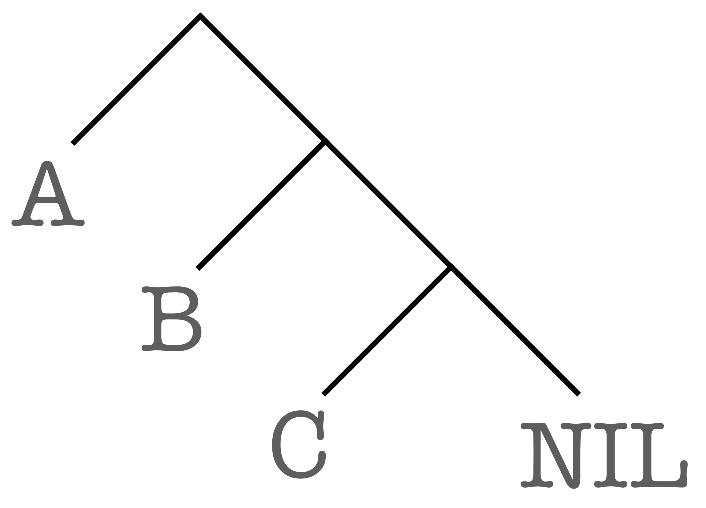
(A B C)

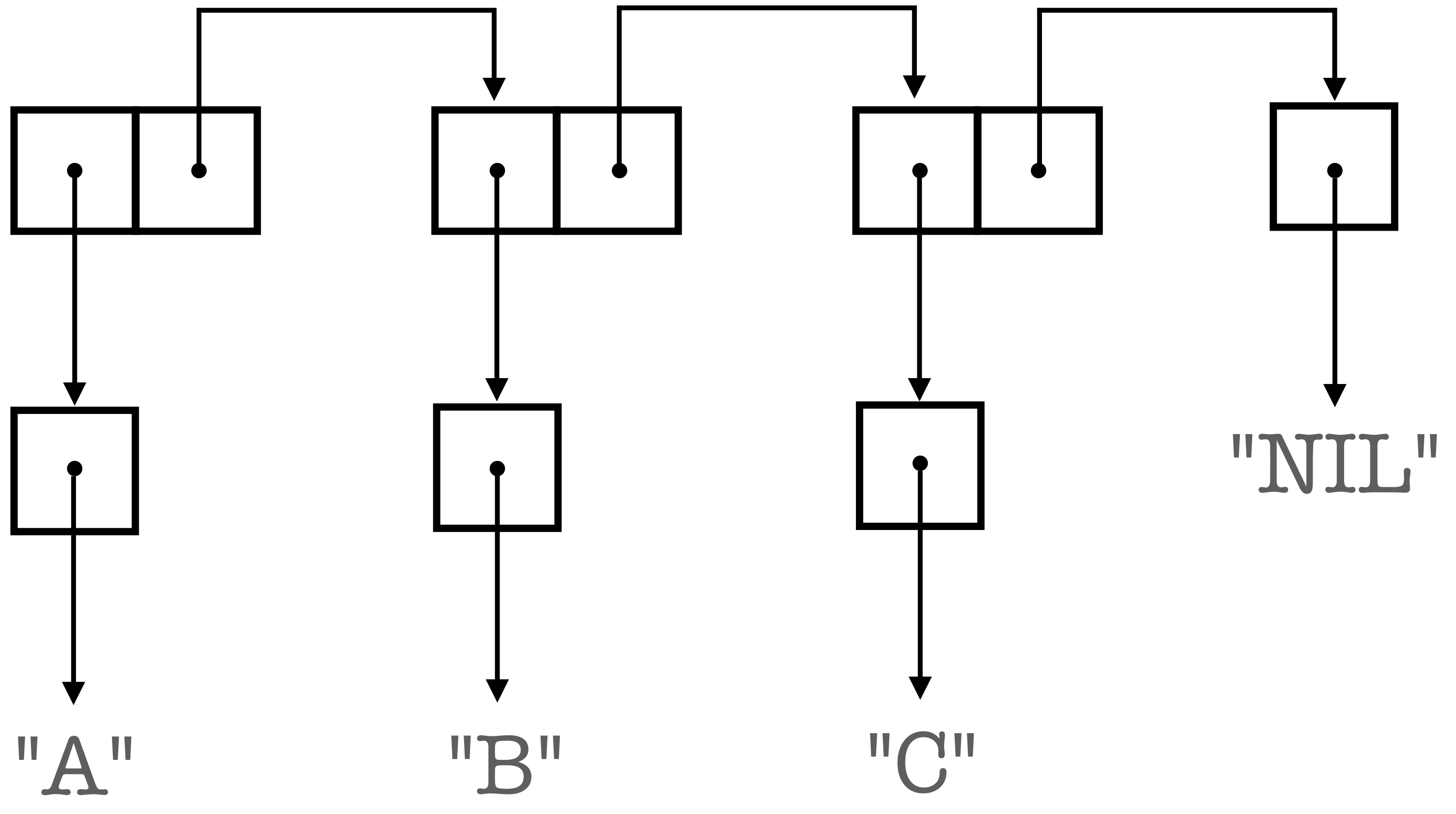




S-Expressions in memory

(A B C)

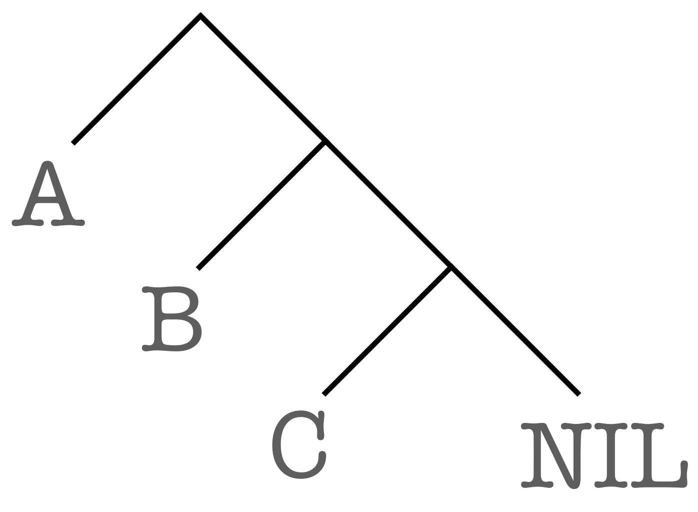


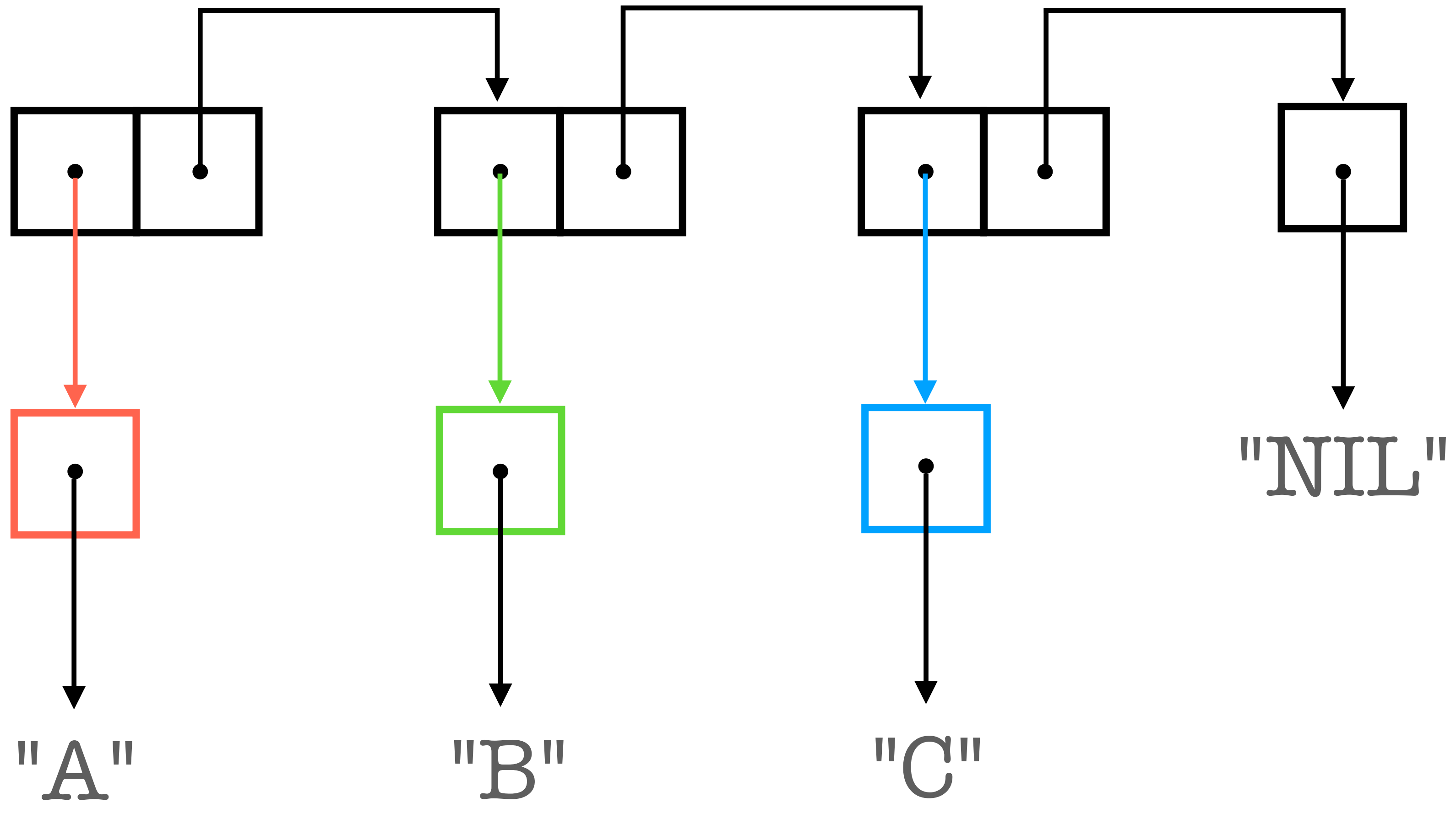


Address	Data	Comment
006000	006014	cons
006002	006004	
006004	006016	cons
006006	006010	
006010	006020	cons
006012	006022	
006014	006025	atom A
006016	006027	atom B
006020	006031	atom C
006022	006035	atom NIL
006024	000101	"A"
006026	000102	"B"
006030	000103	"C"
006032	044516	"NIL"
006034	000114	

S-Expressions in memory

(A B C)

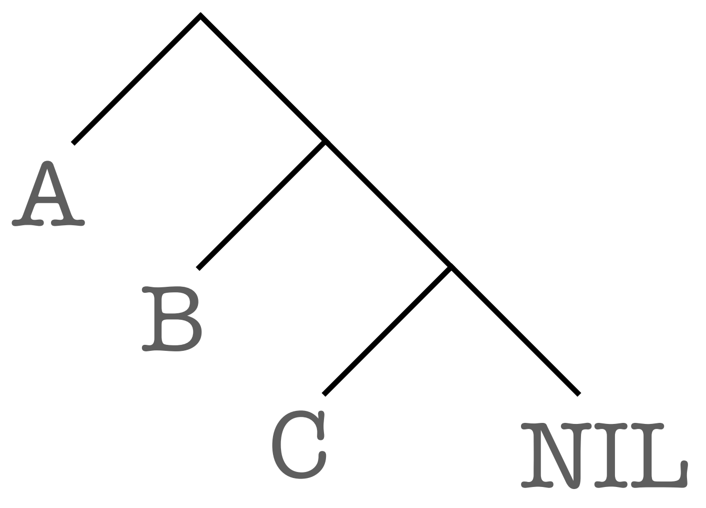


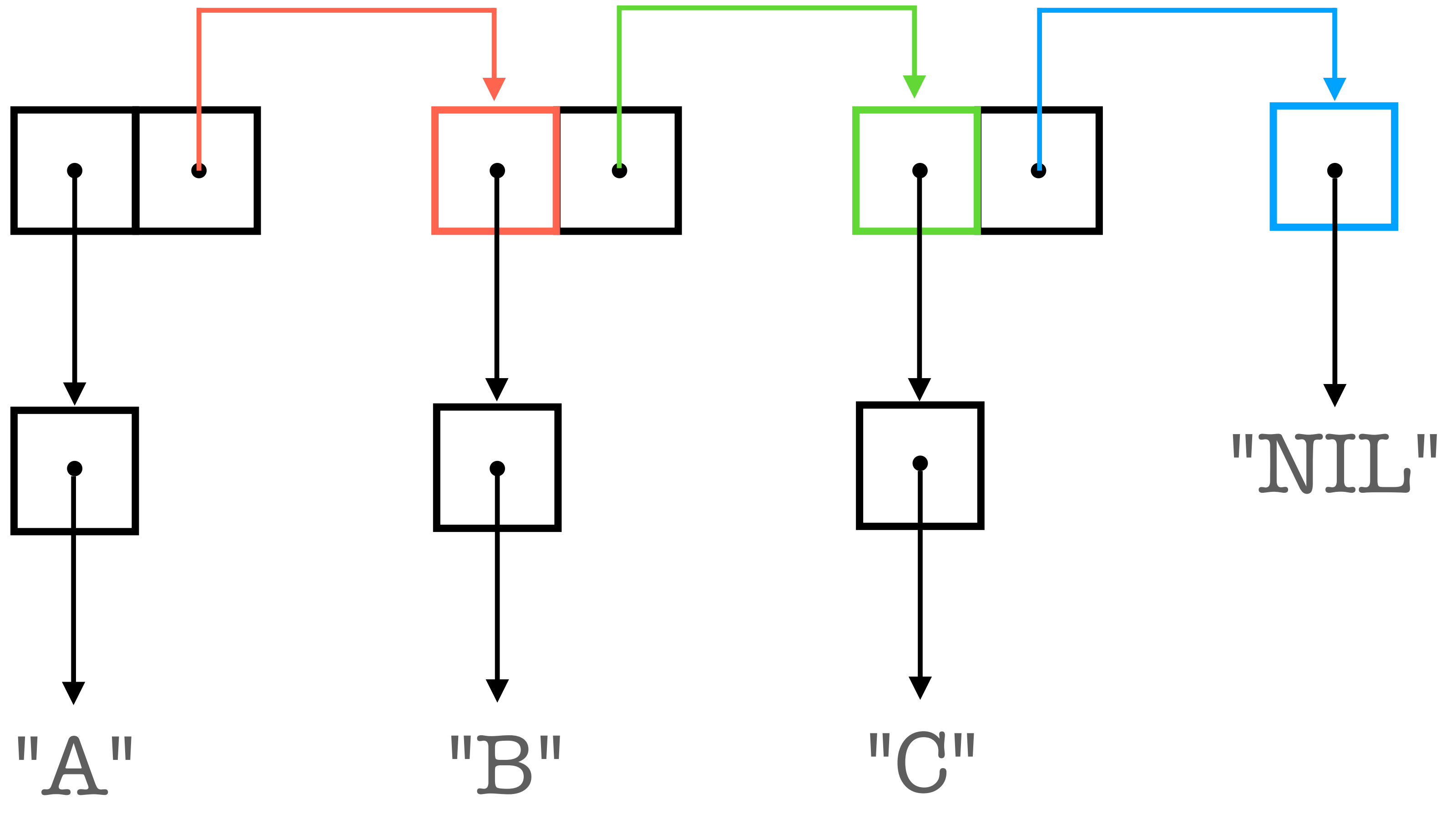


Address	Data	Comment
006000	006014	cons
006002	006004	
006004	006016	cons
006006	006010	
006010	006020	cons
006012	006022	
006014	006025	atom A
006016	006027	atom B
006020	006031	atom C
006022	006035	atom NIL
006024	000101	"A"
006026	000102	"B"
006030	000103	"C"
006032	044516	"NIL"
006034	000114	

S-Expressions in memory

(A B C)

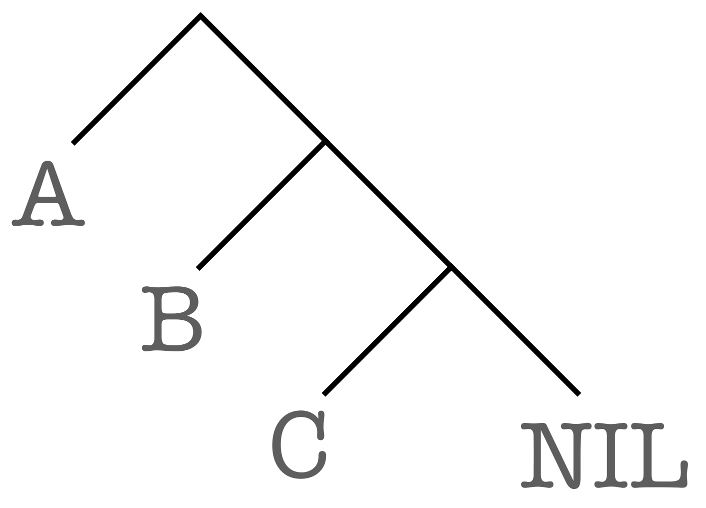


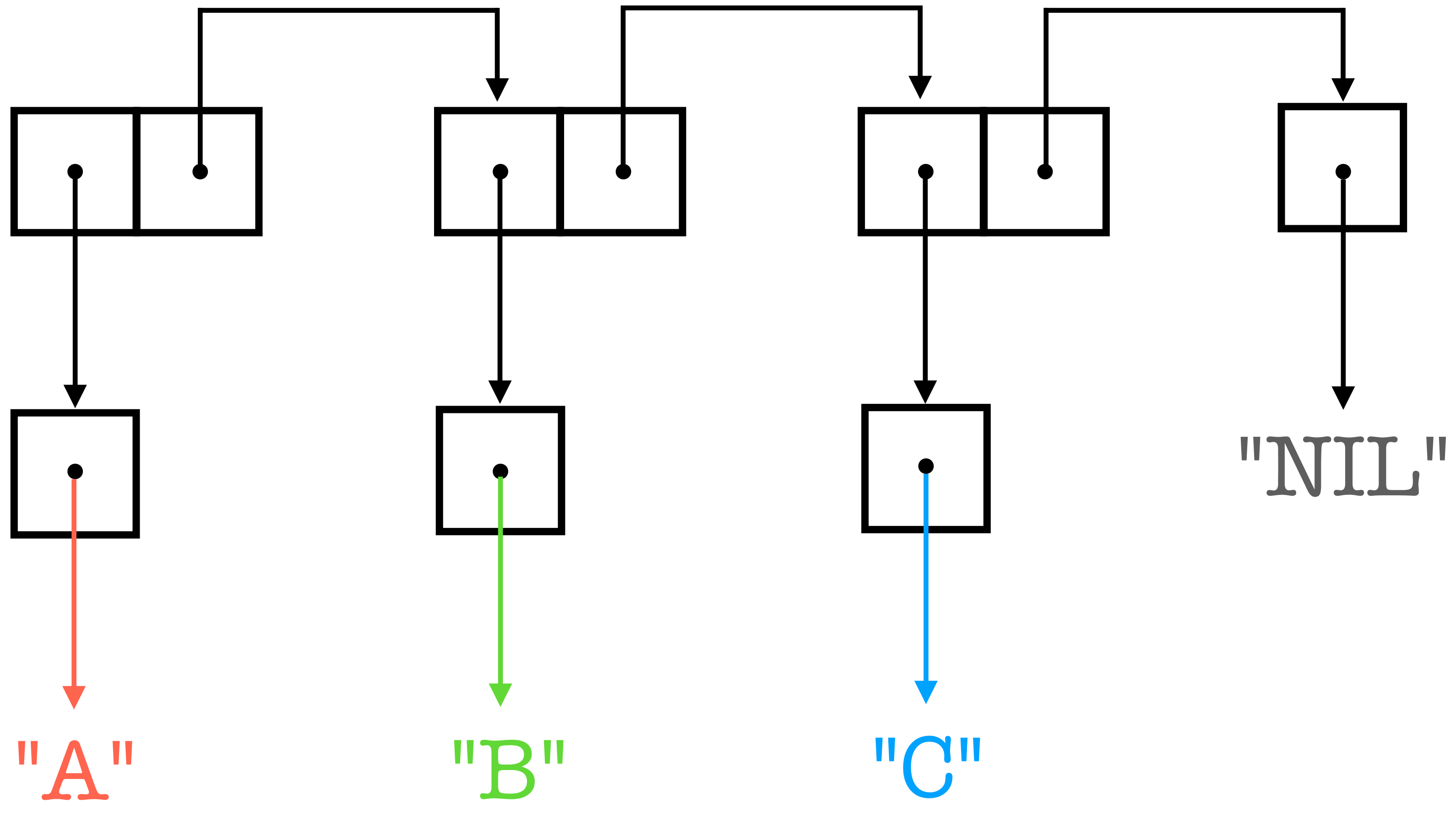


Address	Data	Comment
006000	006014	cons
006002	006004	
006004	006016	cons
006006	006010	
006010	006020	cons
006012	006022	
006014	006025	atom A
006016	006027	atom B
006020	006031	atom C
006022	006035	atom NIL
006024	000101	"A"
006026	000102	"B"
006030	000103	"C"
006032	044516	"NIL"
006034	000114	

S-Expressions in memory

(A B C)





Address	Data	Comment
006000	006014	cons
006002	006004	
006004	006016	cons
006006	006010	
006010	006020	cons
006012	006022	
006014	006025	atom A
006016	006027	atom B
006020	006031	atom C
006022	006035	atom NIL
006024	000101	"A"
006026	000102	"B"
006030	000103	"C"
006032	044516	"NIL"
006034	000114	

REPL overview

- Read
  - Copy input from console
  - Parse input and construct S-Expression
- Eval
  - Evaluate S-Expression
- Print
  - Convert result to string
  - Print to console
- Loop

```
; Init
001000 012706 MOV #1000, SP
001002 001000
001004 005037 CLR @#177560
001006 177560
001010 012737 MOV #10002, @#10000
001012 010002
001014 010000
001016 012705 MOV #6000, R5
001020 006000
001022 000137 JMP @#1100
001024 001100

; REPL
001100 004737 JSR @read
001102 002000
001104 004737 JSR @eval
001106 003000
001110 004737 JSR @print
001112 004000
001114 000137 JMP @#1100 ; loop
001116 001100
```

Primitive S-Functions

- ATOM

```
BIT #1, (R0)
BEQ not_atom
```

Primitive S-Functions

- ATOM
- EQ

```
eq:    MOV @(R0), R0
       DEC R0
       MOV @(R1), R1
       DEC R1

loop:  CMPB (R0), (R1)+
       BNE neq
       TSTB (R0)+
       BEQ done
       BR loop

neq:   CLZ
       RTS PC

done:  SEZ
       RTS PC
```

Primitive S-Functions

- ATOM
- EQ
- CAR

```
MOV @R0, R1
```

Primitive S-Functions

- ATOM
- EQ
- CAR
- CDR

```
MOV 2(R0), R1
```

Primitive S-Functions

- ATOM
- EQ
- CAR
- CDR
- CONS

```
cons: MOV @#10000, R2
      MOV R0, @R2
      MOV R1, 2(R2)
      MOV R2, R0
      ADD 4, R2
      MOV R2, @#10000
      RTS PC
```

Evaluator subroutines

- ATOM, EQ, CAR, CDR, CONS
- QUOTE
- COND
- LAMBDA
- LABEL
- assoc, evlis, evcon

```
assoc: MOV R5, R4
       MOV R0, R2

loop:  R4, #6000
       BLOS bad
       TST -(R4)
       MOV -(R4), R1
       MOV R2, R0
       JSR PC, #eq
       BNE loop
       MOV 2(R4), R0
       RTS PC

bad:   BR bad
```

PDP-11 code layout

Offset	Description
001000	main loop
002000	parser
003000	eval
004000	printer
005000	built-in atoms
006000	symbol table
007000	read/print buffer
010000	heap

```
; Init
001000 012706 MOV #1000, SP
001002 001000
001004 005037 CLR @#177560
001006 177560
001010 012737 MOV #10002, @#10000
001012 010002
001014 010000
001016 012705 MOV #6000, R5
001020 006000
001022 000137 JMP @#1100
001024 001100

; REPL
001100 004737 JSR @read
001102 002000
001104 004737 JSR @eval
001106 003000
001110 004737 JSR @print
001112 004000
001114 000137 JMP @#1100 ; loop
001116 001100
```

Summary

- Print is about 120 bytes
- Read is about 260 bytes
- Eval is about 520 bytes

```
(QUOTE SUBST)
(QUOTE
  (LAMBDA (X Y Z)
    (COND
      ((ATOM Z)
       (COND
         ((EQ Y Z) X)
         ((QUOTE T) Z)))
      ((QUOTE T)
       (CONS
        (SUBST X Y (CAR Z))
        (SUBST X Y (CDR Z)))))))
```

Summary

- Print is about 120 bytes
- Read is about 260 bytes
- Eval is about 520 bytes
- Nexts steps
  - Error handling

```
(QUOTE SUBST)
(QUOTE
  (LAMBDA (X Y Z)
    (COND
      ((ATOM Z)
       (COND
         ((EQ Y Z) X)
         ((QUOTE T) Z)))
      ((QUOTE T)
       (CONS
        (SUBST X Y (CAR Z))
        (SUBST X Y (CDR Z)))))))
```

Summary

- Print is about 120 bytes
- Read is about 260 bytes
- Eval is about 520 bytes
- Nexts steps
  - Error handling
  - **Implement backspace**

```
(QUOTE SUBST)
(QUOTE
  (LAMBDA (X Y Z)
    (COND
      ((ATOM Z)
       (COND
         ((EQ Y Z) X)
         ((QUOTE T) Z)))
      ((QUOTE T)
       (CONS
        (SUBST X Y (CAR Z))
        (SUBST X Y (CDR Z)))))))
```

Summary

- Print is about 120 bytes
- Read is about 260 bytes
- Eval is about 520 bytes
- Nexts steps
  - Error handling
  - **Implement backspace**
  - Add octal literals

```
(QUOTE SUBST)
(QUOTE
  (LAMBDA (X Y Z)
    (COND
      ((ATOM Z)
       (COND
         ((EQ Y Z) X)
         ((QUOTE T) Z)))
      ((QUOTE T)
       (CONS
        (SUBST X Y (CAR Z))
        (SUBST X Y (CDR Z)))))))
```

Code examples

```
(QUOTE SUBST)
(QUOTE
  (LAMBDA (X Y Z)
    (COND
      ((ATOM Z)
       (COND
         ((EQ Y Z) X)
         ((QUOTE T) Z)))
      ((QUOTE T)
       (CONS
        (SUBST X Y (CAR Z))
        (SUBST X Y (CDR Z)))))))
```

```
(QUOTE T)
(QUOTE T)

(QUOTE F)
(QUOTE F)

(QUOTE NOT)
(QUOTE
  (LAMBDA (P)
    (COND
      (P F)
      (T T))))

(QUOTE AND)
(QUOTE
  (LAMBDA (P Q)
    (COND
      (P Q)
      (T F))))

(QUOTE OR)
(QUOTE
  (LAMBDA (P Q)
    (COND
      (P T)
      (T Q))))
```

```
(QUOTE AMONG)
(QUOTE
  (LAMBDA (X Y)
    (COND
      ( (NOT (ATOM Y))
        (COND
          ((EQUAL X (CAR Y)) T)
          (T (AMONG X (CDR Y)))))
      (T F))))
```

```
(QUOTE EQUAL)
(QUOTE
  (LAMBDA (X Y)
    (COND
      ( (AND (ATOM X) (ATOM Y))
        (EQ X Y))
      ( (AND (NOT (ATOM X)) (NOT (ATOM Y)))
        (AND
          (EQUAL (CAR X) (CAR Y))
          (EQUAL (CDR X) (CDR Y))))
      (T F))))
```

Code examples

```
(QUOTE SUBST)
(QUOTE
  (LAMBDA (X Y Z)
    (COND
      ((ATOM Z)
       (COND
         ((EQ Y Z) X)
         ((QUOTE T) Z)))
      ((QUOTE T)
       (CONS
        (SUBST X Y (CAR Z))
        (SUBST X Y (CDR Z)))))))
```

```
(SUBST
  (QUOTE (AN S EXPRESSION))
  (QUOTE AWESOME)
  (QUOTE (THIS IS AWESOME)))
```

```
(THIS IS (AN S EXPRESSION))
```

Questions?