CS 311 Computational Structures

Scheme

Profs Tim Sheard and Andrew Black

Overview

- Scheme is a Turing-complete programming language
- Scheme uses the same structures to represent both programs and data: *S-expressions*
- Scheme has a simple semantics, based upon the lambda calculus
- Scheme is expressive enough to write a universal machine.
- We will write such a machine in the guise of an interpreter for Scheme written in Scheme

History

- Lisp is the second-oldest programming language still in current use.
  - Invented by John McCarthy in 1958
  - Published in *Comm. ACM* in April 1960
- 2 current dialects:
  - Common Lisp
  - Scheme

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,

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

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

Scheme

- Scheme developed at MIT by Guy L. Steele and Gerald Jay Sussman, 1975–1980
- Scheme has a “minimalist” design
  - Small core language + mechanisms for extension
- Defined in a *de facto* standard called the *Revised^n Report on the Algorithmic Language Scheme* (R*n*RS).
  - The most widely implemented standard is R5RS

S-expressions

- S-exps have a simple grammar

  S $\rightarrow$ Atomic | List

  List $\rightarrow$ (Items)

  Items $\rightarrow$ S Items | Λ

  Atomic $\rightarrow$ Symbol | Number | String | Boolean | Procedure

  Symbol $\rightarrow$ symbol i.e. 'X 'yz 'w34

  Number $\rightarrow$ 12.5

  String $\rightarrow$ "abnc" i.e. things inside double quotes

  Boolean $\rightarrow$ #t | #f

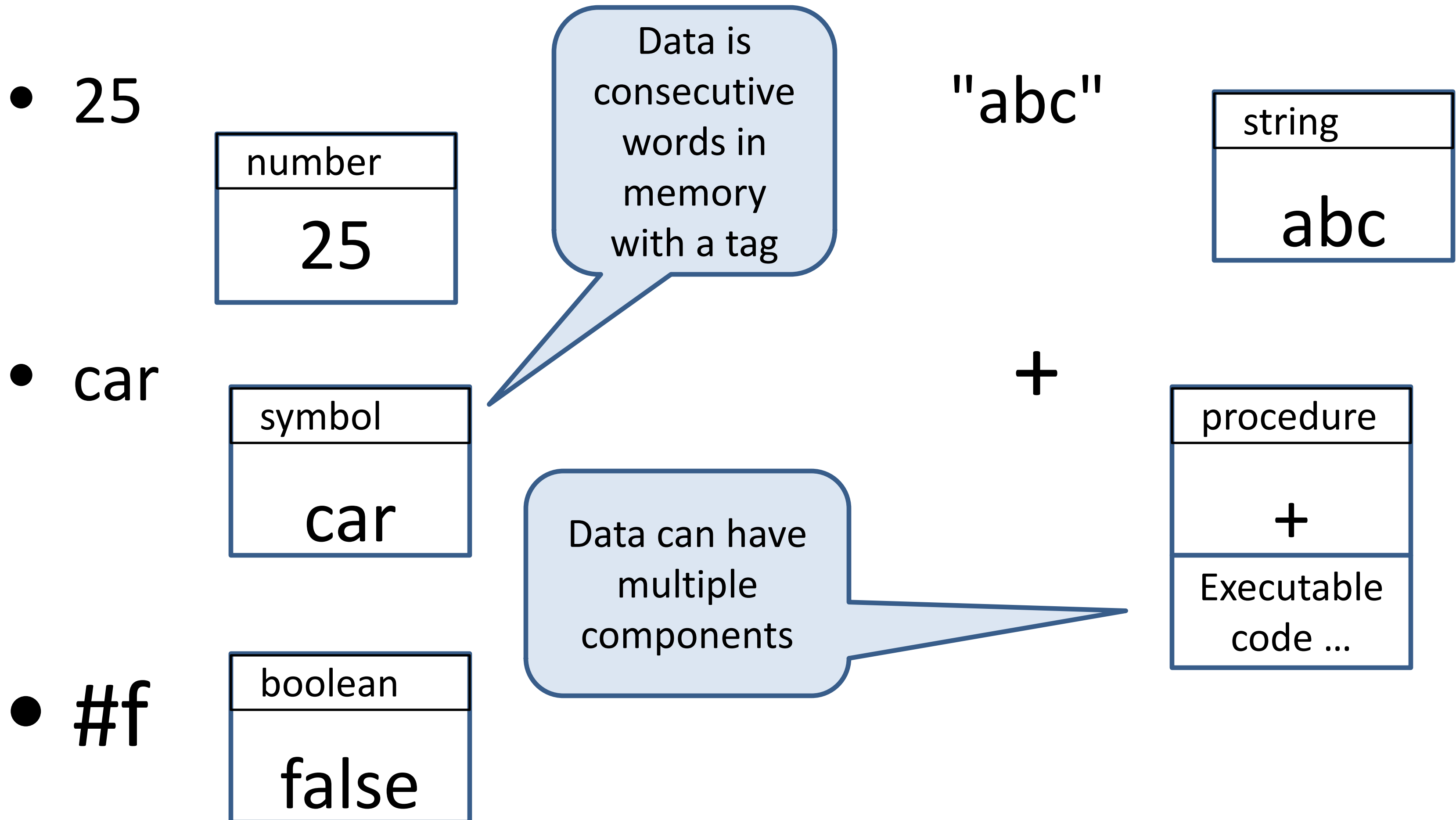
  Procedure $\rightarrow$ primitive built in code

Example S-exp

- 25
- car
- “abc”
- (cdr y)
- (cons 3 5)
- (1 4 tom 7 8)
- (if (> 3 x) 22 y)
- (quote (car x))

Representing atomic data

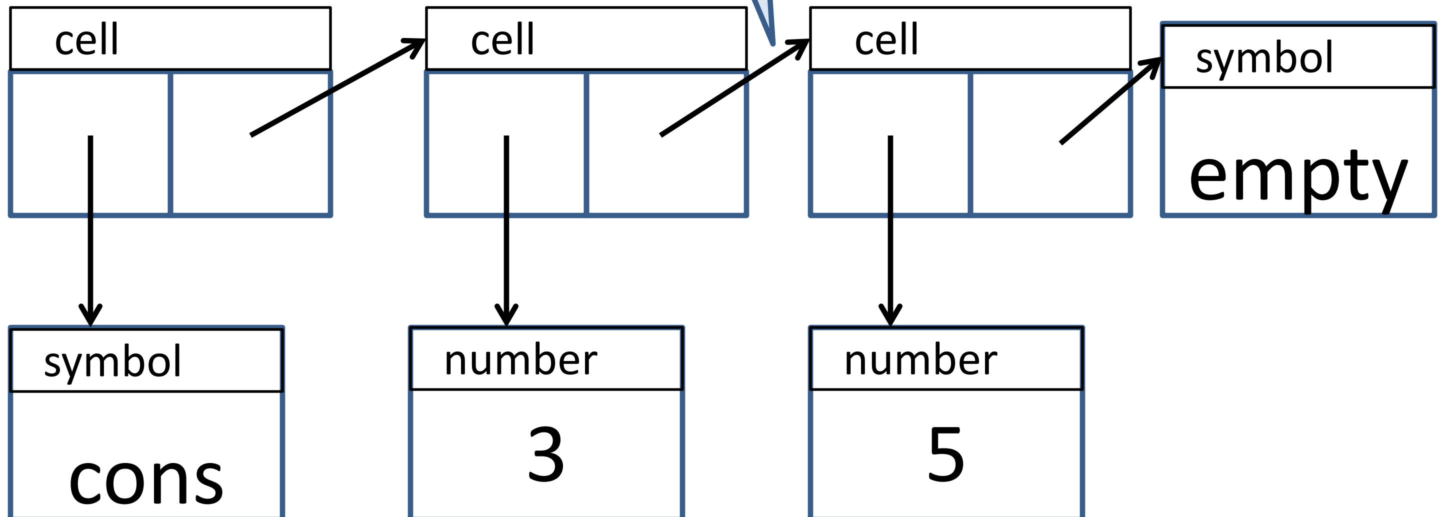
- Atomic data: data that can’t be broken into parts



List data

(cons 3 5)

List are comprised of "cons cells" and pointers



S-expressions as programs

- Data: "abc", 25
- Function calls: “polish notation”
  (+ 3 7) (< (- x y) 23) (reverse x)
- Control structures: (if (< x 5) (g t) (+ 2 x))
- Declarations: (define pi (/ 22.0 7.0))

Programs *vs.* data

- By default, an S-exp is usually interpreted as a program.
- Programs are (usually) "call-by-value":
  - Every element of the S-exp is e*valu*ated
  - The first element is assumed to be a function, which is then *call*ed with the remaining elements as arguments
    - (+ (+ 2 5) 2) $\rightarrow$ (+ 7 2) $\rightarrow$ 9
- S-Exps can also be interpreted as data by *quoting*

Quoting

- If a quote precedes an S-exp, as in '(+ 2 3) or '(4 5 6), then the S-exp following the quote is interpreted as data.
- A quoted S-exp evaluates to itself

- '(a b q) is shorthand for (quote (a b c))

  > '(+ 3 4)

Function calls

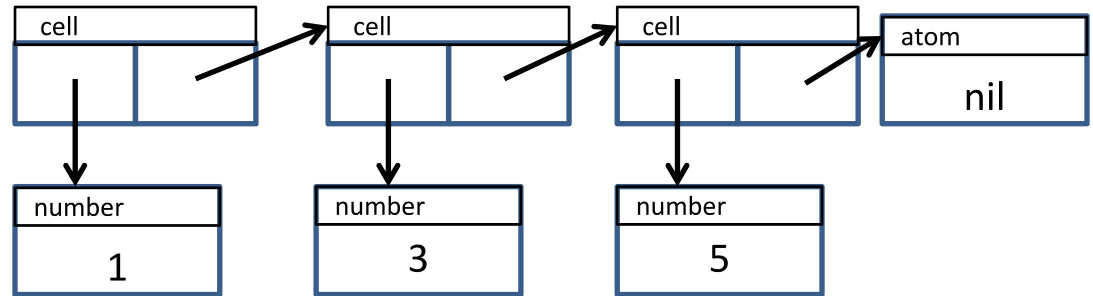
- (h z y z)
  h $\rightarrow$ procedure
  z $\rightarrow$ value
  y $\rightarrow$ value
  z $\rightarrow$ value

- The procedure is then applied to all the arguments

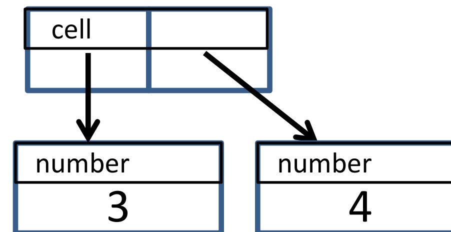
```
(g 2 3)
(f 1)
```

Constructing Data

- Quoting
  '(1 3 5)



- Dot expressions
  '(3 . 4)



- Cons
  (cons 3 (cons 4 '())) → '(3 4)

- List
  (list (+ 3 4) 'abc 9) → '(7 abc 9)

Equality

- Primitive equality on atomic data
  - (eq? x y) are x and y the same object?
- Structural equality
  - (equal? x y) do x and y have the same structure?

```
> (eq? 'a 'a)
#t
> (eq? '(1 2) '(1 2))
#f
>(equal? '(1 2) '(1 2))
#t
```

Symbols are unique

- All Symbols with the same name are represented inside the machine by the same object.

```
> (eq? 'abc 'abc)
#t
> (eq? "abc" (string-append "ab" "c"))
#f
>(eq? 'abc
    (string->symbol (string-append
                    "ab" "c")))
#t
```

Control structures

- quote
- lambda
- cond
- define
- These are part of the language, not functions

- Conditionals
  - (cond (($test_1$ exp_1)($test_2$ exp_2) …))

Definitions

```
(define x 25)

(define (f x) (+ x 9))

(define g +)
```

Anonymous functions

- In Scheme, one can write anonymous functions (*i.e.*, functions without a name)
  - **(lambda (**$param_1$ … $param_n$**)** *body***)**
- Lambdas evaluate to procedures

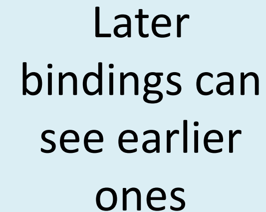
```
> (lambda (x y) (+ (+ x 1) y))
#<procedure>
> ((lambda (x y) (+ (+ x 1) y))  5 7)
13
```

Local binding

- let   and   let*

```
(define (f x)
      (let ((y 1)
           (z 3))
          (+ x y z)))

(define (f x)
      (let* ((y 1)
            (z (+ y 3)))
           (+ x y z)))t*
```



Lexical scoping

- Free variables in anonymous functions see the closest enclosing definition.

```
(define x 25)

(define y (let ((x 1))
            ((lambda (z)
                 (+ z x))
             12)))
```

Functions as arguments

- In Scheme, one can write functions that take other functions as arguments.

```
(define (app1 f y)
   (cond ((symbol? y) (f y))
         ((list? y) (cons (f (car y))
                          (cdr y)))))

> (app1 symbol? 'a)
#t
> (app1 symbol? '(a b c))
(#t b c)
```

Important higher order functions

- map
- apply

```
>(map (lambda (x) (+ x 3))
     '(1 2 3))
(4 5 6)
>(apply string-append
       '("abc" "zyz"))
"abczyz"
```

Summary

- Scheme is a simple language without much syntax
- Programs and data have the same representation
- One can write programs that build and manipulate programs
- Functions and arguments are treated the same (f x y z)