Introduction to Scheme

Gul Agha
CS 421
Fall 2006

How do you describe them

- A language is described by specifying its syntax and semantics
- **Syntax**: The rules for writing programs.
  - We will use Context Free Grammars.
- **Semantics**: What the programs mean.
  - Informally, using English
  - Operationally, showing an interpreter
  - Other: Axiomatic, Denotational, ...
- Look at R5RS for how this is done formally

Context Free Grammars

A (context-free) grammar consists of four parts:
1. A set of **tokens** or **terminals**. These are the atomic symbols in the language.
2. A set of **non-terminals**. These represent the various concepts in the language
3. A set of rules called **productions** mapping the non-terminals to groups of tokens.
4. A **start** non-terminal

Example: English Language

- **Terminals**
- **Non-Terminals**
- **Productions**
- **Main Non-Terminal**

**Example: English Language**

- **Terminals**
  - Words of the language
- **Non-Terminals**
- **Productions**
- **Main Non-Terminal**

**Example: English Language**

- **Terminals**
  - Words of the language
- **Non-Terminals**
  - sentences, verbs, nouns, adverbs, complex, noun phrases, ...
- **Productions**
- **Main (Start) Non-Terminal**
Example: English Language

- **Terminals**
  - Words of the language

- **Non-Terminals**
  - sentences, verbs, nouns, adverbs, complex, noun phrases, ...

- **Productions**
  - Any guesses

- **Main Non-Terminal**
  - sentence

Backus Naur Form

- BNF non-terminals are enclosed in the symbols < ... >
- < > represents the empty string
- Example:
  
  ```
  <string> ::= < > | <char><string>
  <char> ::= a | b | c | ....
  ::= should be read as can be
  | should be read as or
  Sometimes -> is used for ::= 
  ```

A Grammar for Core Scheme

```
<expression> →
  <constant>
  | <identifier>
  | (if <expression> <expression> <expression>)
  | (define <identifier> <expression>)
  | quote <expression>
  | (begin <expression> ... <expression>)
  | (lambda <formals> <body>)
  | <procedure call>
```

Grammar continued

```
<constant> → <boolean>
  | <number>
  | <character>
  | <string>

<formals> → (<variable>*)
  | (<variable>...<variable>...<variable>)
  | <variable>

<body> → <expression>

<procedure call> → (<operator> <operand>*)
<operator> → <expression>
<operand> → <expression>

And productions for <boolean>, <number>, <character>, <string>
```

Syntactic Keywords of Scheme

- **Control:** quote, lambda, if, define, begin.
- **Predicates:** equal? null? zero? ...
- **Recognizers:** integer? boolean? string? pair? list? procedure? ...
- **Lists:** car cdr cons ...

An Overview of Scheme

- **R-E-P**
  - Interacting via the Read-Eval-Print loop
- **Atomic Data**
  - booleans, numbers, symbols, characters & strings ...
- **Expressions**
  - What are they? and how are they evaluated.
- **Conditionals**
  - If and syntactic sugar (macros): not, and, or, cond.
- **Recognizers & Predicates**
Overview of Scheme (Contd.)

- **Functions**
  - lambda, let, let*
- **Definitions**
  - what they are and what they look like.
- **Lists & quote**
  - what they look like, including the empty list
- **List Manipulation**
  - Constructing and deconstructing lists: car, cdr, cons

Recursive Definition

- what it is, some examples, and how it works.

Recursion vs Iteration

- translating a while into a recursive function (tail recursion)

List Processing

- some world famous list functions

Functionals

- functions of function e.g., mapping & sorting

### Binding

- **Pattern Matching**
  - what is it & how is it done.
- **Notions of Equality**
  - eq? and its many relations.
- **Data Mutation & Scheme Actors**
  - setcar!, setcdr! and the sieve of erasthones

**Important Note:** No mutations (assignments) may be used in the first couple of MPs

### Functional Language

- Functions are first-class values in Scheme.
- Functions can be used as arguments
- Functions can be returned as values
- Functions can be bound to variables and stored in data structures

Scheme is not a strictly functional language: bindings of variables can be mutated. We will avoid this for now.

### Talking to Scheme

```scheme
> 3434 3434
> (integer? 2345) #t
> (+ 1 2 3 4) 10
> (define + *)
> (+ 1 2 3 4) 24
> (+) #<primitive-procedure *>
> (lambda (x) (* x x))
> (lambda (a1) ...)
> #<CLOSURE (x) (* x)> 
> ((lambda (x) (* x x)) 12)
> ((lambda (x) (* x x)) 12)
> 144
```

```scheme
> (define cell (cons + *))

> cell
> (cons + *)
> (define + *)
> (procedure? (car cell))
> true
> ((cdar cell) 5 5)
> 25
```
## Atomic Data

- **Booleans**
  - The truth values: #f #t

- **Numbers**
  - Integer
  - Rational
  - Real
  - Complex number

- **Symbols or identifiers**
  - Used as variables to name values

- **Characters & Strings**
  - #\a; lower case letter a
  - #\space; the preferred way to write a space
  - #\newline; the newline character
  - "this is a typical string"

## Notes

- Booleans, numbers, characters are **constants**.
- Symbols (other than predefined functions) must be given a value.
- All these data types are **disjoint**, i.e. nothing can be both a *this* and a *that*.

## Expressions

- **Constants, Symbols and Data**
- **Lists**
  - Special forms
  - Application of a function to given arguments

\[
\text{<exp>} ::= (\text{<exp>} \text{<exp>} \ldots \text{<exp>})
\]

**Stands for:**

\[
\text{<function application>} ::= (\text{<fun>} \text{<arg1>} \ldots \text{<argn>})
\]

## Examples

\[
> (*) 3 4 (+ 2 3)
\]

60

\[
> (*)
\]

1

\[
> ((\text{lambda} (x) (* x 4)) 4)
\]

16

\[
> ((\text{car} (\text{cons} + *)) 7 3)
\]

10

## Sequencing

\[
\text{(begin \ldots \text{<exp>})}
\]

- Evaluates the expressions \text{<exp>} from left right.
- Returns the value of the rightmost expression.

\[
\text{(begin (write \text{"a"}) (write \text{"b"}) (write \text{"c"}) (+ 2 3))}
\]

- \text{"a"="b"="c"}5
  - Writes "a", "b" and "c" out to the screen, in that order.
  - Evaluates to 5.
Lists

- (list? x)
  - evaluates to #t if and only if x is/names a list.
  - Otherwise it evaluates to #f.
- ( ) ; the empty list
- (null? ( ) )
- (null? x)
- (this is a list with a very deep
  (((((((((((((((end))))))))))))))))))

Lists and Expressions

- Scheme expressions are lists
- Expressions are evaluated
- How to say something is a list rather than an expression?
- Use the special form quote:
  > (quote <expression>)
  - evaluates to the list or scheme object expression.
  - No evaluation of expression takes place.
  - Abbreviated as: `'<expression>'

Examples

>`(+ 1 2)   
>`(+ '(* 3 4) 1 2)  
>(procedure? `'+) 
>(procedure? '+)  
>(symbol? `'+)   
>(symbol? +)

Notes

- Notation Convention
  - Recognizers end with a ?
  - There is also a ! convention -- used for something different
- Constants Convention
  - Constants such as booleans, strings, characters, and numbers do not need to be quoted.
- Quoted Lists
  - A quoted list is treated as a constant. It is an error to alter one.
- Nested Lists
  - Lists may contain lists, which may contain lists, which may contain lists ....

List Manipulation Operations

- cons is the list constructor
- car gets the first element of a list.
  - Its name comes from: Contents of the Address Register.
- cdr gets the rest of a list.
  - Its name comes from: Contents of the Decrement Register.
- Names inherited from the first implementation of Lisp on IBM 704 back in 1958.

Examples

1. (cons 1 ()) evaluates to (1)  
2. (cons 5 '(1 2 3 4)) 
3. (cons '(1 2 3 4) '(5)) 
4. (car `(a b c d))  
5. (cdr `(a b c d))  
6. (caar `((1) 2 3 4)) 
7. (cddr `(a b c d))  
8. (cddddr `(a b c d)) 
9. (cdddddr `(a b c d))
Notes

1. (car ()) is undefined, as is (cdr ()).
2. (pair? <list>) => #t when <list> is non-empty.
3. (pair? ())
4. Any composition of cars and cdrs of length up to length 4 is valid,
   - car cdr
   - caar cdar cadr cddr
   - caaar ......... cddddr

Question: How many operations in this last line?

Lisp Manipulation Axioms

1. (car (cons <a> <d>)) is <a>
2. (cdr (cons <a> <d>)) is <d>
3. If <list> is a list, then
   (cons (car <list>) (cdr <list>))
   is equal? to <list>

List Recursion

- Check if it is a list (list? ...)
- Repeatedly apply cdr till you reach the end of the list
- Use (null? ...) to check if it empty before deconstructing
- Use (cons ...) to build a list
- Use (cdr ...) to walk through list

Think Recursively!

Write a function, F, which when given a list <list> produces a new list, obtained by adding 1 to every number in the <list>

(F '(a 1 b 2 c 3 d 4 e 5))
(F '(a 2 b 3 c 4 d 5 e 6))

Base Case and Induction

- If <list> is the empty list, then what should (F <list>) be?
- If I know what (car <list>) is, and I also know what (F (cdr <list>)) is, do I know what (F <list>) should be?

The Answers

- (F '( )) should just be ( )
- if (car <list>) is a number
  (F <list>) should be
  (cons (+ 1 (car <list>)) (F (cdr <list>)))
- if (car <list>) is not a number
  (F <list>) should be
  (cons (car <list>) (F (cdr <list>)))
**The Code**

```scheme
(define F
  (lambda (l)
    (if (null? l)
        ()
        (if (number? (car l))
            (cons (+ 1 (car l)) (F (cdr l)))
            (cons (car l) (F (cdr l)))))))
```

```scheme
> (F '(a 1 b 2 c 3 d 4 e 5))
(a 2 b 3 c 4 d 5 e 6)
```

**Tracing Procedures**

```scheme
> (trace F)
(F)
> (F '(1 2 3 4))
| (F (1 2 3 4))
| | (F (2 3 4))
| | | (F (3 4))
| | | | (F (4))
| | | | | (2 3 4 5)
```

**Understanding cons**
- `cons` creates a new pair each time it is called!
- `eq?` on pairs returns true if and only if the two pairs are the same pairs.

```scheme
> (define a (cons 3 '()))
> (define b (cons 3 '()))
> (eq? a a) #t
> (eq? a b) #f
> (eq? '() '()) #t
> (eq? (car a) (car b)) #t
```

**Box Diagrams**

<table>
<thead>
<tr>
<th>List notation</th>
<th>Pair notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a b c d)</td>
<td>(a . (b . (c . (d . ()))))</td>
</tr>
<tr>
<td>(a b c d)</td>
<td>(a . ((b . (c . ())). (d . ()))))</td>
</tr>
</tbody>
</table>

**Other List Operations**
- `append` takes two or more lists and constructs a new list with all of their elements:
  ```scheme
  > (append '(a b) '(c d))
  (a b c d)
  > (list '(a b) '(c d))
  ((a b) (c d))
  ```

**Implementing append**
- `append` is syntactic sugar provided by Scheme.
- Here is how it could be defined for two elements:
  ```scheme
  > (define my-app
     (lambda (lis1 lis2)
       (if (null? lis1)
         lis2
         (cons (car lis1) (my-app (cdr lis1) lis2))))))
  > (my-app '(1 2 3) '(4 5 6))
  (1 2 3 4 5 6)
  ```
- Beware when you mutate bindings and pass lists! `lis2` is reused!
Pairs

- Pairs are a derived data type
- Values: (datum . datum)
- Operations: cons, car, cdr, pair?, eq?
- Lists are internally represented as pairs
  - (a b c d) → (a . (b . (c . (d . ()))))
  - (a (b c) d) → (a . ((b . (c . (.))). (d . ()))))
- Box Diagrams: Recall that pairs are represented by boxes with car and cdr fields. LHS of dot is car and RHS is cdr

Pairs can be shared

- (define c '(1 . a))
- (define d (cons 2 c))
- (define e (cons 2 c))
- (eq? d e) #f
- (eq? (cadr d) (cadr e)) #t

Vectors

- Vectors are a derived type
- Provide random access instead of sequential access to elements

```
> (define v1 (vector 1 (+ 1 1)))
v1
#2 (1 2)
> (define v2 (vector v1 v1))
v2
#2 (#0=#2 (1 2))
```

Higher-Order Procedures

Composition of functions. If \( g : A \rightarrow B \) and \( f : B \rightarrow C \) then \( f \circ g : A \rightarrow C \) is defined as
\[
(f \circ g)(x) = f(g(x))
\]

```
> (define compose
  (lambda (f g)
    (lambda (x)
      (f (g x))))))
> ((compose car cdr) '(a b c d))
b
```

Curried Functions

Multiple argument function can be represented by single argument functions.
Let \( f : A \times B \rightarrow C \) Define \( \text{curry} : (A \times B) \rightarrow (A \rightarrow B \rightarrow C) \)
\[
curry(f)(x)(y) = f(x,y)
\]
Curry is a bijection.
Defining Curry

```scheme
(define curry
  (lambda (f)
    (lambda (x)
      (lambda (y)
        (f x y))))))

> (((curry +) 1) 2)
3

(define add1
  ((curry +) 1))

> (add1 4)
5
```

Variable Arity Procedures

Examples: list, vector, string, +, *, ...
Variable arity procedures can be defined as
(lambda formal body)

Formal: A single variable. Scheme binds all arguments into a list
Body: An expression

```scheme
> ((lambda x x) 1 2 3)
(1 2 3)
> (define my-list (lambda x x))
> (my-list 1 2 3 4)
(1 2 3 4)
```

Variable Arity Procedure Example

Define plus which
- complains if 0 arguments,
- takes 1 or 2 arguments,
- ignores more arguments.

```scheme
(define plus
  (lambda lst
    (if (null? lst
      (write "wrong number of arguments")
    (if (null? (cdr lst))
      (car lst)
    (+ (car lst)
      (cadr lst))))))

> (plus)
"wrong number of arguments"

> (plus 1)
1

> (plus 2 3)
5

> (plus 2 3 4)
5
```

Local binding

- More syntactic sugar

```scheme
> (let ((x 11) (y 12)) (+ x y))
33
```

Is equivalent to:

```scheme
(((lambda (x)
  (lambda (y)
    (+ x y))) 11) 12)
```

- More complicated with recursion in the bindings

Example: Finding Roots

- Finding roots of equation \( f(x) = 0 \), where \( f \) is continuous
- Given points \( f(a) < 0 < f(b) \), \( f \) must have at least one zero between \( a \) and \( b \).
- Compute midpoint \( x \) of \( a \) and \( b \), if \( f(x) > 0 \), search between \( f(a) \) and \( f(x) \), otherwise between \( f(x) \) and \( f(b) \).

See, Abelson and Sussman, *The Structure and Interpretation of Computer Programs*.

Search for zero-crossing

(Scheme code)

```scheme
(define search
  (lambda (f neg-point pos-point)
    (let ((midpoint (average neg-point pos-point)))
      (if (close-enough? neg-point pos-point)
        midpoint
      (let ((test-value (f midpoint)))
        (cond ((positive? test-value)
          (search f neg-point midpoint))
          ((negative? test-value)
            (search f midpoint pos-point))
          (else midpoint)))))))

(define close-enough?
  (lambda (x y)
    (< (abs (- x y)) .001)))
```

```scheme
(define search
  (lambda (f neg-point pos-point)
    (let ((midpoint (average neg-point pos-point)))
      (if (close-enough? neg-point pos-point)
        midpoint
      (let ((test-value (f midpoint)))
        (cond ((positive? test-value)
          (search f neg-point midpoint))
          ((negative? test-value)
            (search f midpoint pos-point))
          (else midpoint)))))))

(define close-enough?
  (lambda (x y)
    (< (abs (- x y)) .001)))
```
Example: Derivatives

(An approximation to) the derivative of a function is defined by:

\[
Df(x) = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx}
\]

(define derivative
  (lambda (f dx)
    (lambda (x)
      (/ (- (f (+ x dx)) (f x))
          dx))))

References

- An Introduction to Scheme and its Implementation
  http://www.federated.com/~jim/schintro-v14/schintro_toc.html

- Structure and Interpretation of Computer Programs

- Revised5 Report on the Algorithmic Language Scheme
  http://www.schemers.org/Documents/Standards/R5RS/