1 Background

- Scheme is based on LISP which was developed by John McCarthy in the mid 50s.
- LISP stands for LIST Processing, not Lots of Irritating Silly Parentheses.
- Functions and data share the same representation: S-Expressions.
- A basic LISP implementation needs six functions cons, car, cdr, equal, atom, cond.
- Scheme was developed by Sussman and Steele in 1975.

2 S-Expressions

- An S-Expression is a balanced list of parentheses.

More formally, an S-expression is
1. a literal (i.e., number, boolean, symbol, character, string, or empty list).
2. a list of s-expressions.
3. Literals are sometimes called atoms.

3 S-Expressions — Examples

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>(</td>
</tr>
<tr>
<td>()</td>
<td>(5)</td>
</tr>
<tr>
<td>(4 5)</td>
<td>() ()</td>
</tr>
<tr>
<td>((5))</td>
<td>(4 (5) (6 (7)))</td>
</tr>
<tr>
<td>(())</td>
<td>)</td>
</tr>
</tbody>
</table>
4 S-Expressions as Trees

- An S-expression can be seen as a linear representation of tree-structure:

```
2
/ \
6 3 4

2
/  \
(6)
```

- S-Expressions as Function Calls

A special case of an S-expression is when the first element of a list is a function name.

Such an expression can be evaluated.

```
> (+ 4 5)
9
> (add-five-to-my-argument 20)
25
> (draw-a-circle 20 45)
#t
```

5 S-Expressions as Functions

- As we will see, function definitions are also S-expressions:

```
(define (fahrenheit-celsius f)
  (* (- f 32) 5/9))
```

- So, Scheme really only has one syntactic structure, the S-expression, and that is used as a data-structure (to represent lists, trees, etc), as function definitions, and as function calls.

6 Function Application

- In general, a function application is written like this:

```
(operator arg₁ arg₂ ... argₙ)
```

- The evaluation proceeds as follows:
1. Evaluate operator. The result should be a function \( F \).
2. Evaluate
   \[
   \text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n
   \]
to get
   \[
   \text{val}_1, \text{val}_2, \ldots, \text{val}_n
   \]
3. Apply \( F \) to \( \text{val}_1, \text{val}_2, \ldots, \text{val}_n \).

8 Function Application — Examples

\[
\begin{align*}
> & (\ + \ 4 \ 5) \\
& 9 \\
> & (+ (+ 5 6) 3) \\
& 14 \\
> & 7 \\
& 7 \\
> & (4 5 6) \\
& \text{eval: 4 is not a function} \\
> & \#t \\
& \#t
\end{align*}
\]

9 Atoms — Numbers

Scheme has

- Fractions (5/9)
- Integers (5435)
- Complex numbers (5+2i)
- Inexact reals (#i3.14159265)

\[
\begin{align*}
> & (+ 5 4) \\
& 9 \\
> & (+ (* 5 4) 3) \\
& 23 \\
> & (+ 5/9 4/6) \\
& 1.2 \\
> & 5/9 \\
& 0.5
\end{align*}
\]
10 Atoms — Numbers...

> (+ 5/9 8/18)
1
> 5+2i
5+2i
> (+ 5+2i 3-i)
8+i
> (* 236542164521634 3746573426573425643)
886222587860913289285513763860662
> pi
#i3.141592653589793
> e
#i2.718281828459045
> (* 2 pi)
#i6.283185307179586

11 Atoms — Numbers...

- Scheme tries to do arithmetic exactly, as much as possible.
- Any computations that depend on an inexact value becomes inexact.
- Scheme has many builtin mathematical functions:

> (sqrt 16)
4
> (sqrt 2)
#i1.4142135623730951
> (sin 45)
#i0.8509035245341184
> (sin (/ pi 2))
#i1.0

12 Atoms — Strings

- A string is enclosed in double quotes.

> (display "hello")
hello
> "hello"
"hello"
> (string-length "hello")
5
> (string-append "hello" " world!")
"hello world!"

13 Atoms — Booleans

- true is written #t.
- false is written #f.
<#t
true
<#f
false
< (display #t)
#t
< (not #t)
false

14 Identifiers

- Unlike languages like C and Java, Scheme allows identifiers to contain special characters, such as ! $ % & * + - . / : < = > ? @ ^ ~. Identifiers should not begin with a character that can begin a number.
- This is a consequence of Scheme's simple syntax.
- You couldn’t do this in Java because then there would be many ways to interpret the expression X-5+Y.

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>h-e-l-l-o</td>
<td>3some</td>
</tr>
<tr>
<td>give-me!</td>
<td>-stance</td>
</tr>
<tr>
<td>WTF?</td>
<td></td>
</tr>
</tbody>
</table>

15 Defining Variables

- define binds an expression to a global name:

  (define name expression)

  (define PI 3.14)

  > PI
  3.14

  (define High-School-PI (/ 22 7))

  > High-School-PI
  3.142857

16 Defining Functions

- define binds an expression to a global name:

  (define (name arg₁ arg₂ ...) expression)

- arg₁ arg₂ ... are formal function parameters.
(define (f) 'hello)

> (f)
hello

(define (square x) (* x x))

> (square 3)
9

17 Defining Helper Functions

- A Scheme program consists of a large number of functions.
- A function typically is defined by calling other functions, so called helper or auxiliary functions.

(define (square x) (* x x))
(define (cube x) (* x (square x)))

> (cube 3)
27

18 Preventing Evaluation

- Sometimes you don’t want an expression to be evaluated.
- For example, you may want to think of (+ 4 5) as a list of three elements +, 4, and 5, rather than as the computed value 9.
- (quote (+ 4 5)) prevents (+ 4 5) from being evaluated. You can also write ’(+ 4 5).

> (display (+ 4 5))
9
> (display (quote (+ 4 5)))
(+ 4 5)
> (display ’(+ 4 5))
(+ 4 5)

19 Dr Scheme

- Download DrScheme from here: http://www.drscheme.org.
- It has already been installed for you in lectura and the Windows machines in the lab.
- Start DrScheme under unix (on lectura) by saying

  > drscheme

- On Windows and MacOS it may be enough to click on the DrScheme logo to start it up.
20 Dr Scheme

Welcome to DrScheme, version 208. Language: Advanced Student.

21 Dr Scheme — Using TeachPacks
22 Dr Scheme — Using the Stepper

23 References

- Tutorials:
  - http://cs.wcc.edu/~7Ecs_dept/KU/PR/Scheme.html
  - http://www.cis.upenn.edu/~7Eungar/CIS520/scheme-tutorial.html
- http://dmoz.org/Computers/Programming/Languages/Lisp/Scheme

24 References...


25 Scheme so Far

- A function is defined by

  (define (name arguments) expression)

- A variable is defined by

  (define name expression)

- Strings are inclosed in double quotes, like "this". Common operations on strings are
- (string-length string)
- (string-append list-of-strings)

- Numbers can be *exact integers*, *inexact reals*, *fractions*, and *complex*. Integers can get arbitrarily large.
- Booleans are written #t and #f.

## 26 Scheme so Far...

- An inexact number is written: \#i3.14159265.
- Common operations on numbers are
  - (+ arg1 arg2), (- arg1 arg2)
  - (add1 arg), (sub1 arg)
  - (min arg1 arg2), (max arg1 arg2)
- A function application is written:

  ```scheme
  > (function-name arguments)
  ```

- Quoting is used to prevent evaluation

  ```scheme
  (quote argument)
  ```

  or

  ```scheme
  'argument
  ```