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.

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.

- Literals are sometimes called atoms.

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))</td>
</tr>
<tr>
<td>() ()</td>
<td>)</td>
</tr>
<tr>
<td>((4 5) (6 (7)))</td>
<td></td>
</tr>
</tbody>
</table>
S-Expressions as Trees

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

(2 (3 4) (5 (6) 7))

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

S-Expressions as Functions

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

(define (fahrenheit-2-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.

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 \( arg₁, arg₂, ... argₙ \) to get \( val₁, val₂, ... valₙ \).
3. Apply \( F \) to \( val₁, val₂, ... valₙ \).
Function Application — Examples

> (+ 4 5)
9
> (+ (+ 5 6) 3)
14
> 7
7
> (4 5 6)
eval: 4 is not a function
> #t
#t

Atoms — Numbers

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

> (+ 5 4)
9
> (+ (* 5 4) 3)
23
> (+ 5/9 4/6)
1.2
> 5/9
0.5

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 built-in mathematical functions:

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

- A string is enclosed in double quotes.

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

Atoms — Booleans

- true is written #t.
- false is written #f.

```
> #t
true
> #f
false
> (display #t)
#t
> (not #t)
false
```

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>

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`
Defining Functions

- `define` binds an expression to a global name:
  
  ```scheme```
  (define (name arg1 arg2 ...) expression)
  ```scheme```

  `arg1 arg2 ...` are formal function parameters.

  ```scheme```
  (define (f) 'hello)
  ```scheme```

  > (f)
  hello

  ```scheme```
  (define (square x) (* x x))
  ```scheme```

  > (square 3)
  9

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.

  ```scheme```
  (define (square x) (* x x))
  ```scheme```

  ```scheme```
  (define (cube x) (* x (square x)))
  ```scheme```

  > (cube 3)
  27

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 `(quote (+ 4 5))`.

  ```scheme```
  > (display (+ 4 5))
  9
  ```scheme```

  ```scheme```
  > (display (quote (+ 4 5)))
  (+ 4 5)
  ```scheme```

  ```scheme```
  > (display '(+ 4 5))
  (+ 4 5)
  ```scheme```

Dr Scheme

- Download DrScheme from here: [http://www.drscheme.org](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
  ```scheme```
  > drscheme
  ```scheme```

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

Dr Scheme — Using TeachPacks

Dr Scheme — Using the Stepper

References

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


Scheme so Far...

- A function is defined by
  
  \( (\text{define } (\text{name } \text{arguments}) \text{ expression}) \)

- A variable is defined by
  
  \( (\text{define } \text{name } \text{expression}) \)

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

- Numbers can be exact integers, inexact reals, fractions, and complex. Integers can get arbitrarily large.

- Booleans are written #t and #f.

Scheme so Far...

- An inexact number is written: \#i3.14159265.

- Common operations on numbers are
  
  - \( (+ \text{arg1 arg2}), (- \text{arg1 arg2}) \)
  - \( (\text{add1 arg}), (\text{sub1 arg}) \)
  - \( (\text{min arg1 arg2}), (\text{max arg1 arg2}) \)

- A function application is written:
  
  \( > (\text{function-name arguments}) \)

- Quoting is used to prevent evaluation
  
  \( (\text{quote argument}) \)

  or

  \( '\text{argument} \)