CSc 520

Principles of Programming Languages

3: Scheme — Introduction

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
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>
An S-expression can be seen as a linear representation of tree-structure:

\[
\begin{array}{c|c|c}
2 & 6 & (6) \\
3 & 4 & (3) \quad (4) \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
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:

\[(\text{operator } \text{arg}_1 \text{ arg}_2 \ldots \text{ arg}_n)\]

The evaluation proceeds as follows:

1. Evaluate \text{operator}. The result should be a function \(\mathcal{F}\).
2. Evaluate \arg_1, \arg_2, \ldots, \arg_n to get \val_1, \val_2, \ldots, \val_n.
3. Apply \(\mathcal{F}\) to \val_1, \val_2, \ldots, \val_n.
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 (\(\frac{5}{9}\))
- 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...

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

  ```scheme
  > (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`.

```scheme
> #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>

[15]
Defining Variables

(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:
  
  \[
  (\text{define} \ (\text{name} \ arg_1 \ arg_2 \ \ldots) \ \text{expression})
  \]

- \(arg_1\ arg_2\ \ldots\) are **formal function parameters**.

  \[
  (\text{define} \ (f) \ 'hello)
  \]

  \[
  > \ (f)
  \]

  hello

  \[
  (\text{define} \ (\text{square} \ x) \ (* \ x \ x))
  \]

  \[
  > \ (\text{square} \ 3)
  \]

  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.

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

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

```
> (cube 3)
27
```
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.

\((\text{quote} \ (+ \ 4 \ 5))\) prevents \((+ \ 4 \ 5)\) from being evaluated. You can also write \'(\(+ \ 4 \ 5)\).

\[
\begin{align*}
> & \ (\text{display} \ (+ \ 4 \ 5)) \\
& 9 \\
> & \ (\text{display} \ \text{(quote} \ (+ \ 4 \ 5))) \\
& (+ \ 4 \ 5) \\
> & \ (\text{display} \ ’(+ \ 4 \ 5)) \\
& (+ \ 4 \ 5)
\end{align*}
\]
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
  
  > drscheme

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

> (start 300 300)
true
> (draw-circle (make-posn 50 50) 20)
true
> |

Select a Teachpack

arrow-gui.ss
arrow.ss
circle.ss
draw.ss
docs.ss
draw2.ss
elevator.ss
graphing2.ss
graphing.ss
muvee.ss

[22]
Dr Scheme — Using the Stepper

```
(define (f2c f)
  (* (- f 32) 5/9))
(define (c2f c)
  (+ 32 (* c 9/5)))
(c2f (f2c 32))
```

Welcome to DrScheme
Language: Intermediate

```
(define (f2c f) (* (- f 32) 0.5))
(define (c2f c) (+ 32 (* c 1.8)))
(((lambda (c)
    (+ 32 (* c 1.8)))
  (f2c 32))
((lambda (f)
    (*
      (- f 32)
      0.5)))
  32))
```

4.2
Read/Write
not running
References

- Tutorials:
  - http://cs.wwc.edu/~Ecs_dept/KU/PR/Scheme.html
  - http://dmoz.org/Computers/Programming/Languages/Lisp/Scheme
References...

- Language reference manual:
  [http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps](http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps).

- Some of this material is taken from
Scheme so Far

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

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

- 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.
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
- \((\text{quote \ argument})\)
- \('\text{argument}'\)