CSc 520

Principles of Programming Languages

34 : Scheme — Introduction

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

Copyright © 2008 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.

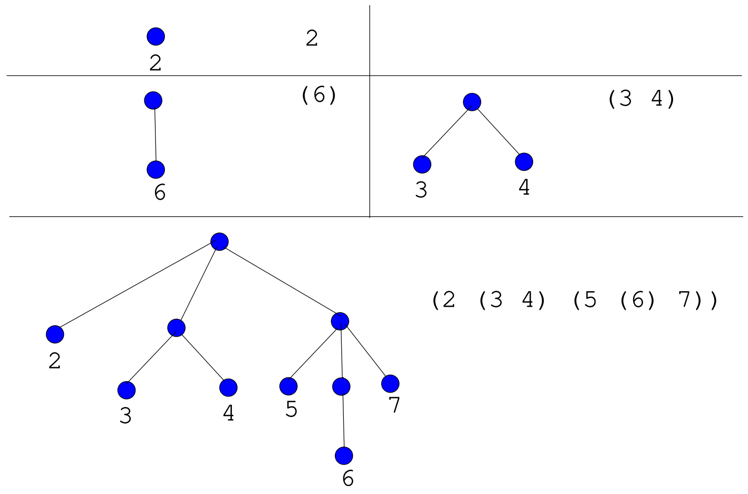
- Literals are sometimes called atoms.

S-Expressions — Examples

Legal	Illegal
`66`	`(`
`()`	`(5))`
`(4 5)`	`()()`
`((5))`	`(4 (5)`
`(()())`	`)(`
`((4 5) (6 (7)))`	

S-Expressions as Trees

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



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 (farenheit-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:

$$(\texttt{operator}\ \texttt{arg}_1\ \texttt{arg}_2\ \ldots\ \texttt{arg}_n)$$

- The evaluation proceeds as follows:
  1. Evaluate `operator`. The result should be a function $\mathcal{F}$.
  2. Evaluate

     $$\texttt{arg}_1,\ \texttt{arg}_2,\ \ldots\ \texttt{arg}_n$$

     to get

     $$\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$$

  3. Apply $\mathcal{F}$ to $\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$.

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 (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:

```
> (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`.

Legal	Illegal
`h-e-l-l-o`	`3some`
`give-me!`	`-stance`
`WTF?`	

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:

  (define (name arg_1 arg_2 ...) *expression*)

- arg_1 arg_2 ... are formal function parameters.

```
(define (f) 'hello)

> (f)
hello

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

> (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
```


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)
```

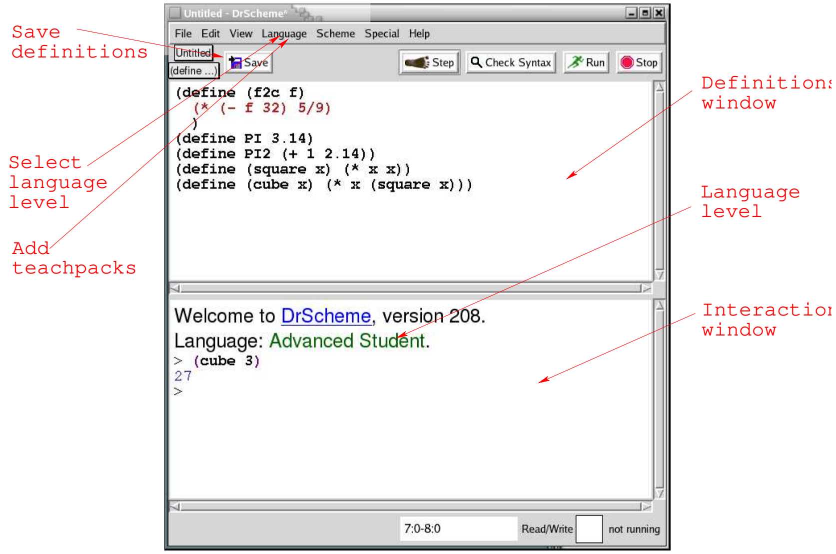

Dr Scheme

- 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

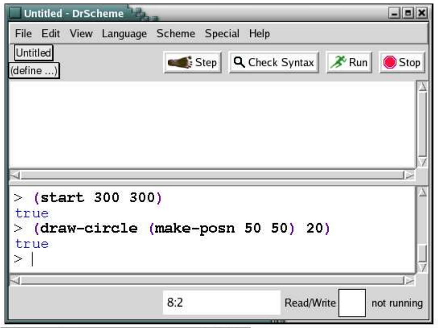
  ```
  > drscheme
  ```

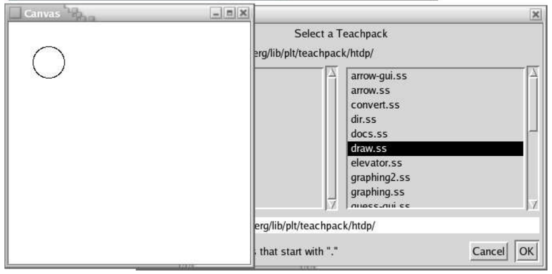
- On Windows and MacOS it may be enough to click on the DrScheme logo to start it up.

Dr Scheme

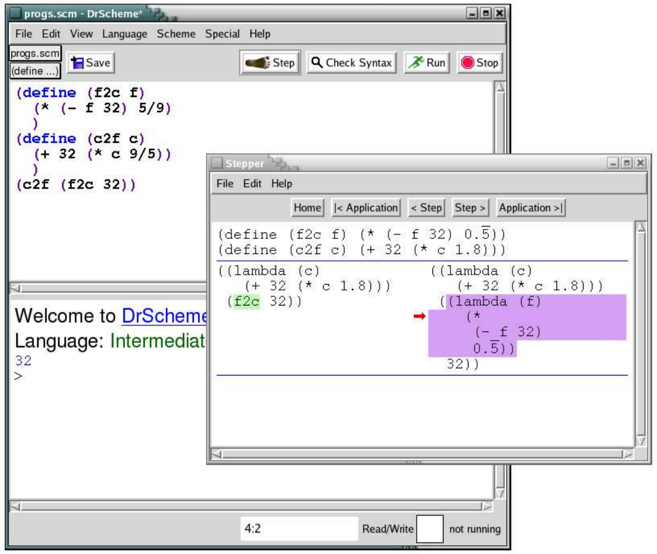


Dr Scheme — Using TeachPacks





Dr Scheme — Using the Stepper



References

- Read Scott, pp. 523-527, 528-539.
- Free interpreter: http://www.drscheme.org.
- Manual:
  http://www.swiss.ai.mit.edu/projects/scheme/documentation/scheme.html
- Tutorials:
  - http://ai.uwaterloo.ca/~dale/cs486/s99/scheme-tutorial.html
  - http://cs.wwc.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

References…

- Language reference manual:
  http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps.
- Some of this material is taken from
  http://www.ecf.utoronto.ca/~gower/CSC326F/slides, ©Diana Inkpen 2002, Suzanne Stevenson 2001.

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`.

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:

  `> (function-name arguments)`

- Quoting is used to prevent evaluation

  `(quote argument)`

  or

  `'argument`