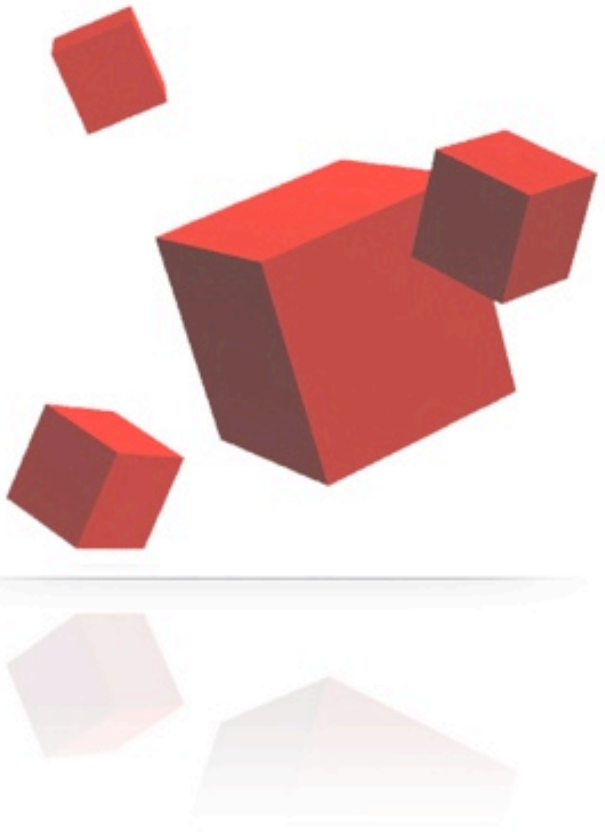
Functional Languages Interpretation in Smalltalk



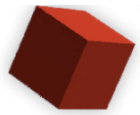
Alexandre Bergel
Lero & DSG, Trinity College Dublin, Ireland

Alexandre.Bergel@cs.tcd.ie
www.cs.tcd.ie/Alexandre.Bergel

November 2006

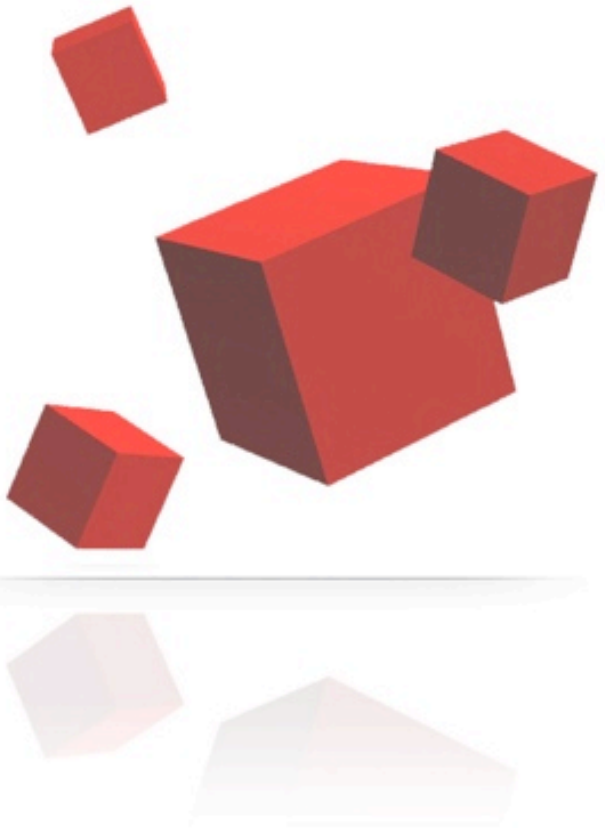
Goal

- Studying the implementation of Scheme-like languages in an object-oriented environment.
- Dynamic versus static scoping



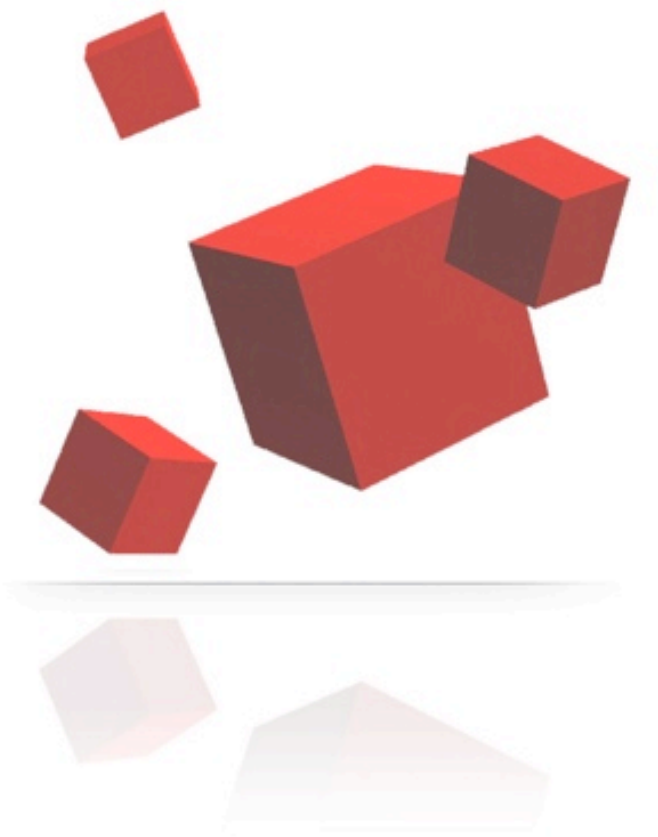
Outline

1. Essence of Scheme
2. SmallScheme overview
3. Modeling environment
4. Reading and parsing Scheme
5. Primitive expression types
6. Dynamic and static scoping
7. Further notes...



Essence of Scheme

1. Idea behind Scheme
2. Examples
3. Evaluation

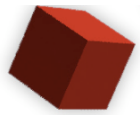


Essence of Scheme

"Scheme demonstrates that a very small number of rules for forming expression, with no restriction on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today."

R5RS, page 2

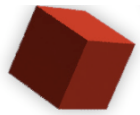
The first description of Scheme was written in 1975



Manipulate parenthesized expressions

Example of Scheme expression:

```
10                          => 10
'aSymbol                    => aSymbol
(+ 1 2)                     => 3
(if cond else then)
(let ((a 10)
      (b 15)
   (+ a (* 2 b)))           => 40
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
(fac 10)                    => 3628800
```

Closure application

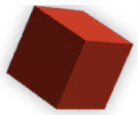
In such expression:

```
(expr expr1 expr2 ... exprN)
```

1 - `expr` is first evaluated into `f`

2 - `expr1 ... exprN` are evaluated (most of the time sequentially). Let's assume results are `v1 ... vN`

3 - `v1 ... vN` are applied to `f`



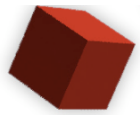
Special form (1/3)

Exceptions to the previous rule are special forms.

Condition of a **`if`** statement needs to be evaluated before other arguments.

```
(if (= x 0)
  'error
  (/ 1 x))
```

`(= x 0)` is evaluated first, then, according to this value, `'error` or `(/ 1 x)` is evaluated.

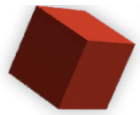


Special form (2/3)

The body of a function is evaluated only when the function is applied

```
(define (foo x) (+ x (* 2 x)))
```

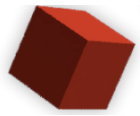
It binds `(lambda (x) (+ x (* 2 x)))` to `foo`



Special form (3/3)

The body of an anonymous function is evaluated only when applied

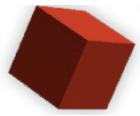
```
((lambda (x) (* x x)) 2)
```

Manipulate parenthesized expressions

Sequence of evaluation

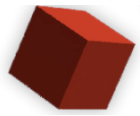
```
(fac 2)
```

Manipulate parenthesized expressions

Sequence of evaluation

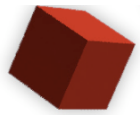
```
(fac 2)
((lambda (n) (if (= n 0)
                 1
                 (* n (fac (- n 1))))) 2)
```

Manipulate parenthesized expressions

Sequence of evaluation

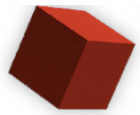
```
(fac 2)
((lambda (n) (if (= n 0)
                 1
                 (* n (fac (- n 1))))) 2)
(if (= 2 0)
    1
    (* 2 (fac (- 2 1))))
```

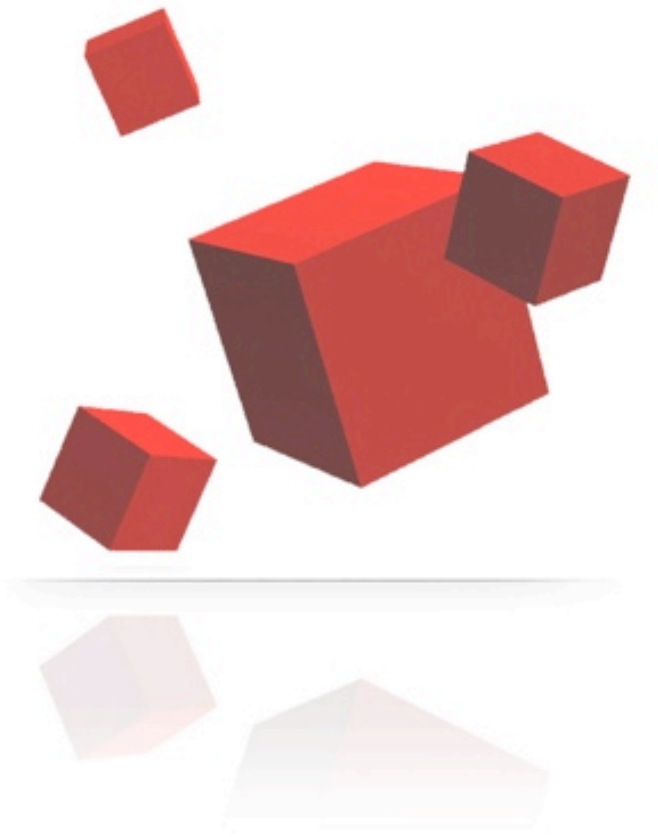
Manipulate parenthesized expressions

Sequence of evaluation

```
(fac 2)
((lambda (n) (if (= n 0)
                 1
                 (* n (fac (- n 1))))) 2)
(if (= 2 0)
    1
    (* 2 (fac (- 2 1))))

(* 2 (fac (- 2 1)))
(* 2 (fac (- 2 1)))
(* 2 (fac (- 2 1)))
(* 2 ((lambda (n) (if (= n 0) ...)) (- 2 1)))
...
```

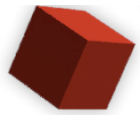



SmallScheme overview



What is SmallScheme

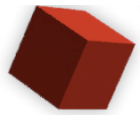
- SmallScheme is an interpreter of a subset of Scheme in Squeak.



How to use SmallScheme

In a programmatic way, Scheme expression can be evaluated:

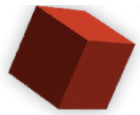
```
Scheme evalString: '(+ 2 3)'
=> 5

Scheme evalString:
   '(define (fac n)
      (if (= n 0)
          1
          (* n (fac (- n 1)))))
     (fac 10)'
=> 3628800
```

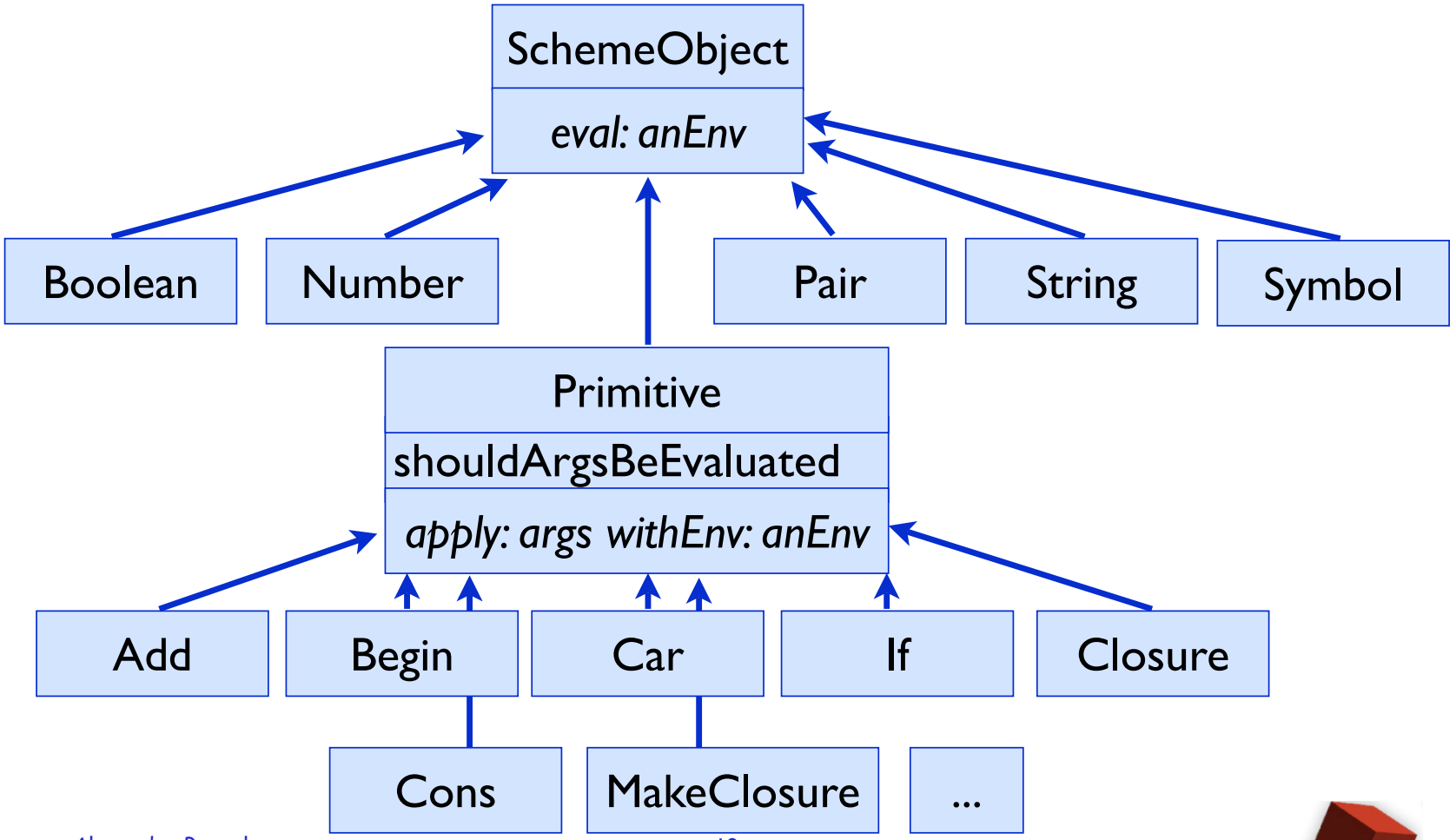
Writing Scheme in Smalltalk with SmallScheme

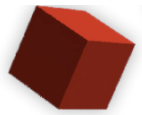
Available on www.squeaksource.com

```
MCHttpRepository
  location: 'http://www.squeaksource.com/Scheme'
  user: ''
  password: ''
```

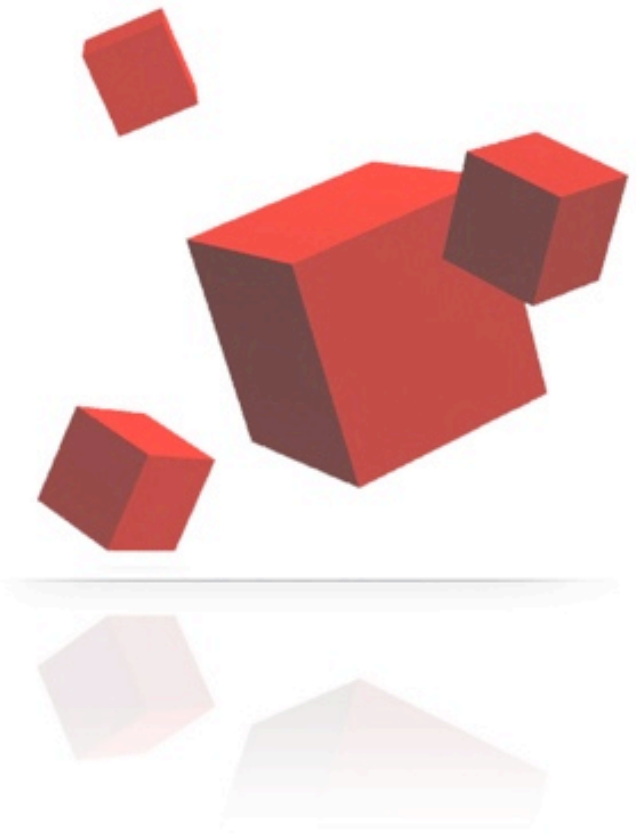
Class Hierarchy





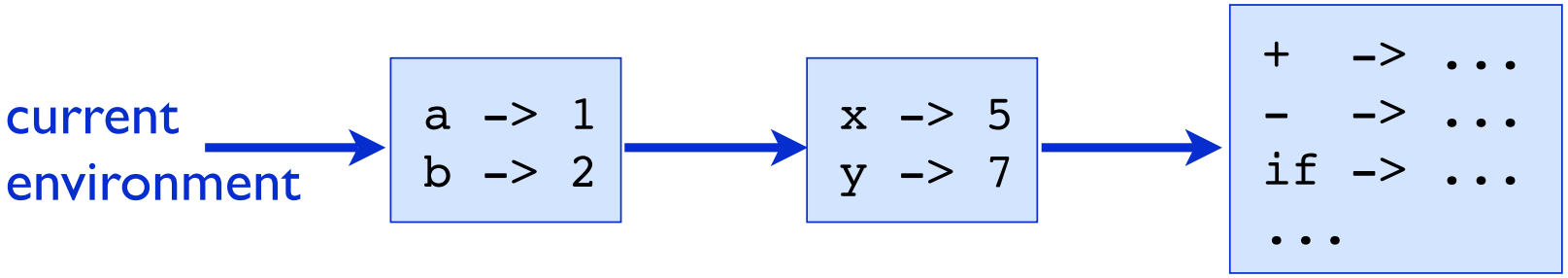
Modeling environment

1. Set of bindings
2. Hierarchy of environments
3. Primitive definitions

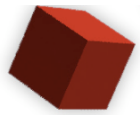


Environment

```
(let ((x 5) (y 7))
  (let ((a 1) (b 2))
    ...))
```



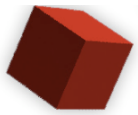
An environment has a parent and a set of bindings



Modeling Environments (1/3)

Each environment has a set of bindings and a parent:

```
Object subclass: #SchemeEnvironment
  instanceVariableNames: 'dictionary parent'

SchemeEnvironment>>get: aSymbol
  ^ dictionary
     at: aSymbol asSmalltalkObject
     ifAbsent: [parent get: aSymbol]

SchemeEnvironment>>at: aSymbol put: aValue
  dictionary
     at: aSymbol asSmalltalkObject
     put: aValue
```

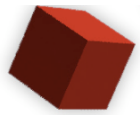



Modeling Environment (2/3)

Creating a new environment:

```
SchemeEnvironment class>>createFromEnv: anEnv
  ^ self new withEnvironment: anEnv.

SchemeEnvironment>>withEnvironment: anEnv
  parent := anEnv.
  dictionary := Dictionary new.
  parent isNil
     ifTrue: [self initializePrimitives].
```

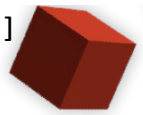
The environment root should be aware of different primitives.



Modeling Environment (3/3)

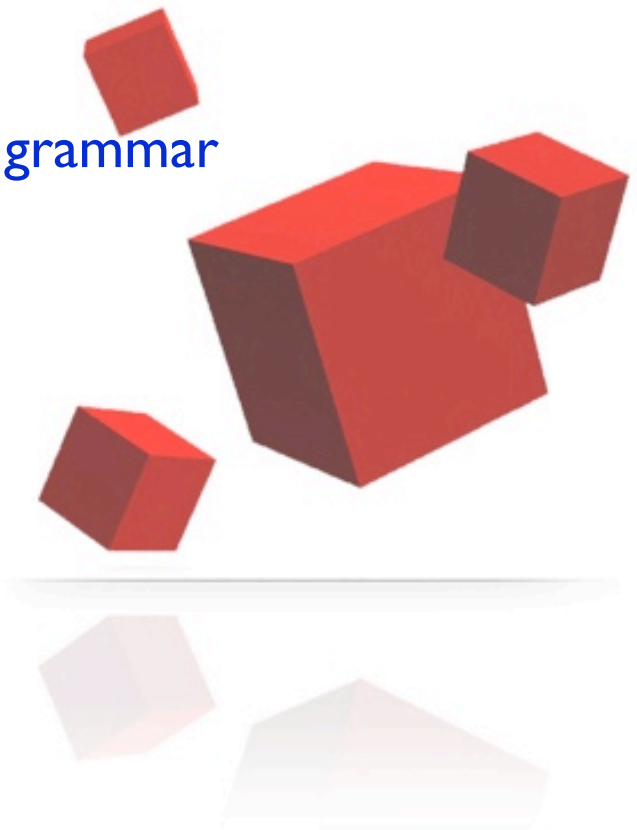
The root of the environment contains the primitives:

```
SchemeEnvironment>>initializePrimitives
    {{'if' . SchemePrimitiveIf} .
    {'begin' . SchemePrimitiveBegin} .
    {'loadfile' . SchemePrimitiveLoadfile} .
    {'display' . SchemePrimitiveDisplay} .
    {'newline' . SchemePrimitiveNewline} .
    {'set!' . SchemePrimitiveSet} .
    {'let' . SchemePrimitiveLet} .
    {'>' . SchemePrimitiveGreater} .
    {'<' . SchemePrimitiveLesser} .
    {'+' . SchemePrimitiveAdd} .
    {'/' . SchemePrimitiveDiv} .
    {'*' . SchemePrimitiveMul} .
    {'-' . SchemePrimitiveSub} .
    {'=' . SchemePrimitiveEqual} .
    {'define' . SchemePrimitiveDefine} .
    {'quote' . SchemePrimitiveQuote} .
    {'lambda' . SchemePrimitiveMakeClosure} .
    {'null?' . SchemePrimitiveNull} .
    {'car' . SchemePrimitiveCar} .
    {'cdr' . SchemePrimitiveCdr} .
    {'cons' . SchemePrimitiveCons}}
       do: [:pair| self at: pair first asSymbol put: pair second new]
```



Reading Scheme (R5RS, Section 1.2, 2, 7)

1. Lexical analysis
2. Parsing Scheme
3. SmallScheme uses a minimal grammar



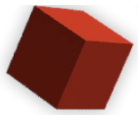
Lexical Analysis

Defined by the class `SchemeLexer`

Public methods are:

```
SchemeLexer class>>analyseString: aString
SchemeLexer >>next
SchemeLexer >>getToken
SchemeLexer >>getType
```

The type of a token could either be a boolean, (,), '(), an integer, a string, or a symbol.



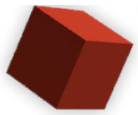
Parsing Scheme

Defined by the class `SchemeParser`

Public methods are:

```
SchemeParser class>>lexer: aLexer
SchemeParser >>next
```

The `next` method returns a scheme object



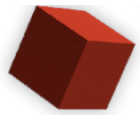
Grammar of SmallScheme

Very minimal grammar

```
expr =
    <number>        |
    <symbol>        |
    <string>        |
    <nil>           |
    ( expr+ )       |
    ‘ expr
```

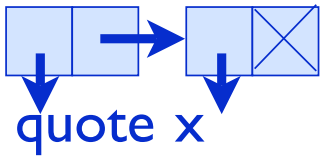
Primitive types are not part of the grammar. This gives more flexibility regarding the definition of those types.

This is a major difference between SmallScheme and R5RS.

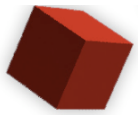


Example: the quote case

‘x = (quote x) =

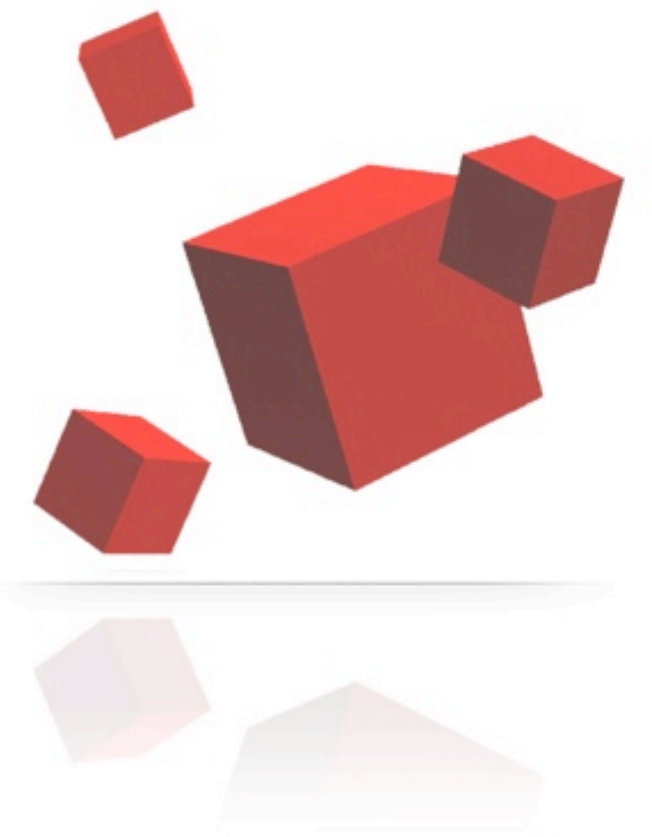



```
SchemeParser>>next
   | l |
   l := lexer next.
   ...
   lexer getType == SchemeLexer quoteID
     ifTrue: [
        ^ SchemePair
           car: (‘quote’ asSchemeSymbol)
           cdr: (SchemePair
                    car: self next
                    cdr: nil)]
```

Primitive expression types (R5RS, Section 4.1)

1. Variable reference
2. Literal expression (quote)
3. Procedure calls
4. Procedures (lambda)
5. Conditionals

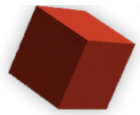


Variable Reference

Class `SchemeSymbol`:

```
SchemeObject subclass: #SchemeSymbol
  instanceVariableNames: 'value'

SchemeSymbol>>eval: anEnvironment
  ^anEnvironment get: value
```

Scheme evalString: '(define x 5) x'
=> 5

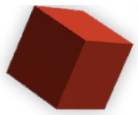


Literal expression (quote)

Class `SchemePrimitiveQuote`:

```
SchemePrimitive subclass: #SchemePrimitiveQuote

SchemePrimitiveQuote>>
  apply: listOfArguments withEnv: anEnvironment
  "arguments are not evaluated"
  ^listOfArguments car

SchemePrimitiveQuote>>initialize
  self doNotEvaluateArguments
```

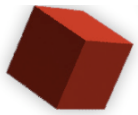
Scheme evalString: '(quote (+ 1 2)'
=> (+ 1 2)



Procedure calls

Associated with the `lambda` keyword

```
SchemePrimitive subclass:
            #SchemePrimitiveMakeClosure

SchemePrimitiveMakeClosure>>
  apply: listOfArguments withEnv: anEnvironment

    |args body|
    args := listOfArguments car.
    body := listOfArguments cdr.

    ^ SchemePrimitiveClosure
       createWithArgs: args
       body: body
       env: anEnvironment
```

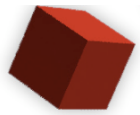
Procedure calls

```
SchemePrimitive subclass: #SchemePrimitiveClosure
   instanceVariableNames: 'arguments body env'
```

When we execute the following:
Scheme evalString: '((lambda (x y) (+ x y)) 2 3)'

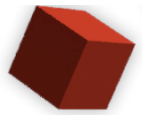
```
SchemePrimitiveClosure>>
   apply: listOfArguments withEnv: anEnvironment
   ...
```

`listOfArguments` is a collection containing 2 and 3



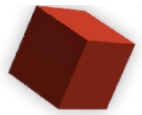
Procedure calls

```
SchemePrimitiveClosure>>
   apply: listOfArguments withEnv: anEnvironment
   | env b valuesProvided argNames |
   valuesProvided := listOfArguments.
   argNames := arguments.
   env := SchemeEnvironment createFromEnvironment: environment.

   [valuesProvided isNil] whileFalse: [
      env at: argNames car put: valuesProvided car.
      argNames := argNames cdr.
      valuesProvided := valuesProvided cdr.
      ].

   b := body.

   [b cdr isNil] whileFalse: [
      b car eval: env.
      b := b cdr].

   ^ b car eval: env
```

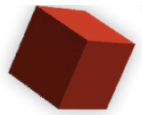
Creation of a new environment

```
SchemePrimitiveClosure>>
   apply: listOfArguments withEnv: anEnvironment
   | env b valuesProvided argNames |
   valuesProvided := listOfArguments.
   argNames := arguments.
   env := SchemeEnvironment createFromEnvironment: environment.

   [valuesProvided isNil] whileFalse: [
      env at: argNames car put: valuesProvided car.
      argNames := argNames cdr.
      valuesProvided := valuesProvided cdr.
      ].

   b := body.

   [b cdr isNil] whileFalse: [
      b car eval: env.
      b := b cdr].

   ^ b car eval: env
```

Binding variables in the new environment

```
SchemePrimitiveClosure>>
   apply: listOfArguments withEnv: anEnvironment
   | env b valuesProvided argNames |
   valuesProvided := listOfArguments.
   argNames := arguments.
   env := SchemeEnvironment createFromEnvironment: environment.

   [valuesProvided isNil] whileFalse: [
      env at: argNames car put: valuesProvided car.
      argNames := argNames cdr.
      valuesProvided := valuesProvided cdr.
      ].

   b := body.

   [b cdr isNil] whileFalse: [
      b car eval: env.
      b := b cdr].

   ^ b car eval: env
```

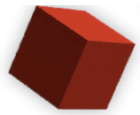



Evaluating the body of the closure

```
SchemePrimitiveClosure>>
   apply: listOfArguments withEnv: anEnvironment
   | env b valuesProvided argNames |
   valuesProvided := listOfArguments.
   argNames := arguments.
   env := SchemeEnvironment createFromEnvironment: environment.

   [valuesProvided isNil] whileFalse: [
      env at: argNames car put: valuesProvided car.
      argNames := argNames cdr.
      valuesProvided := valuesProvided cdr.
      ].
```

```
b := body.

[b cdr isNil] whileFalse: [
   b car eval: env.
   b := b cdr].
```

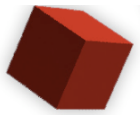
```
   ^ b car eval: env
```

And returning the last value...

```
SchemePrimitiveClosure>>
   apply: listOfArguments withEnv: anEnvironment
   | env b valuesProvided argNames |
   valuesProvided := listOfArguments.
   argNames := arguments.
   env := SchemeEnvironment createFromEnvironment: environment.

   [valuesProvided isNil] whileFalse: [
      env at: argNames car put: valuesProvided car.
      argNames := argNames cdr.
      valuesProvided := valuesProvided cdr.
      ].

   b := body.

   [b cdr isNil] whileFalse: [
      b car eval: env.
      b := b cdr].
```

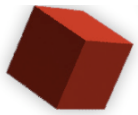
```
^ b car eval: env
```

Conditionals

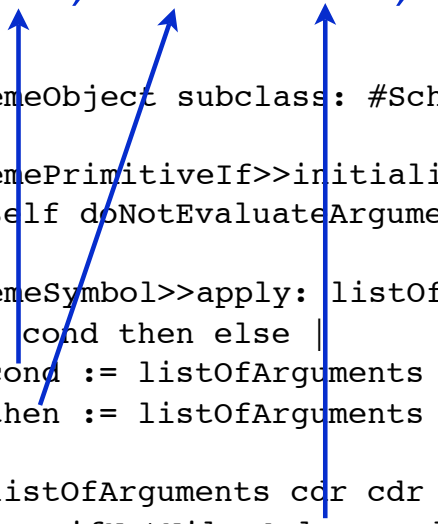
(if (= 2 3) ‘hello ‘world)

```
SchemeObject subclass: #SchemePrimitiveIf

SchemePrimitiveIf>>initialize
   self doNotEvaluateArguments

SchemeSymbol>>apply: listOfArguments withEnv: anEnvironment
   | cond then else |
   cond := listOfArguments car.
   then := listOfArguments cdr car.

   listOfArguments cdr cdr
        ifNotNil: [else := listOfArguments cdr cdr car].

   cond := cond eval: anEnvironment.
   cond isTrue
        ifTrue: [^ then eval: anEnvironment].
   else notNil
        ifTrue: [^ else eval: anEnvironment].
   ^ Scheme undefined
```

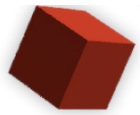



Conditionals explained

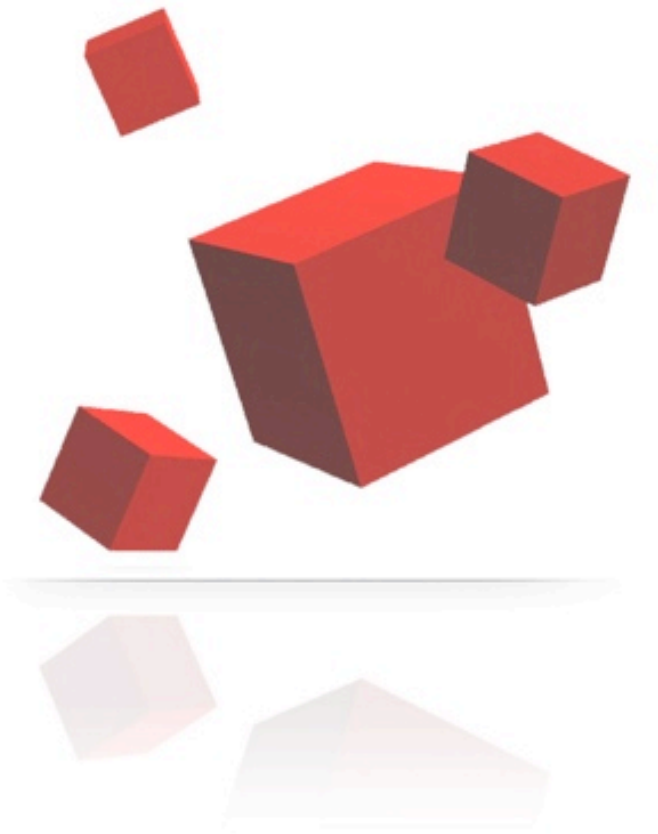
(if (= 2 3) 'hello 'world)



```
SchemeObject subclass: #SchemePrimitiveIf

SchemePrimitiveIf>>initialize
   self doNotEvaluateArguments

SchemeSymbol>>apply: listOfArguments withEnv: anEnvironment
   | cond then else |
   cond := listOfArguments car.
   then := listOfArguments cdr car.

   listOfArguments cdr cdr
        ifNotNil: [else := listOfArguments cdr cdr car].

   cond := cond eval: anEnvironment.
   cond isTrue
        ifTrue: [^ then eval: anEnvironment].
   else notNil
        ifTrue: [^ else eval: anEnvironment].
   ^ Scheme undefined
```

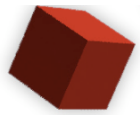



Dynamic and static scoping



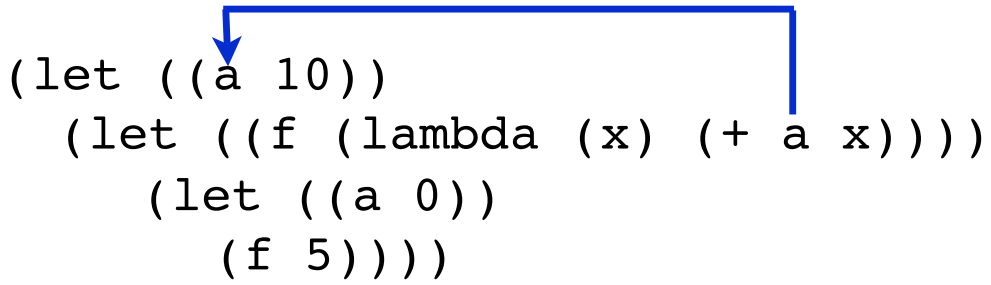
Dynamic vs static scoping

- *Static scoping:* variable references are bound to the environment in which those variables are defined.
- *Dynamic scoping:* variables are looked up dynamically, according to the current environment

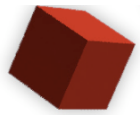


Static scoping: variable are statically bound

The result is 15



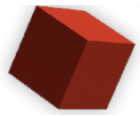
```
(let ((a 10))
  (let ((f (lambda (x) (+ a x))))
     (let ((a 0))
        (f 5))))
```

Dynamic scoping: variable are dynamically bound

The result is 5

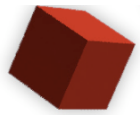
```
(let ((a 10))
  (let ((f (lambda (x) (+ a x))))
     (let ((a 0))
        (f 5))))
```

Static scoping with SmallScheme

```
SchemePrimitiveClosure>>
   apply: listOfArguments withEnv: anEnvironment
   | env b valuesProvided argNames |
   valuesProvided := listOfArguments.
   argNames := arguments.
   env := SchemeEnvironment createFromEnvironment: environment.
   ...
```

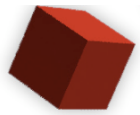
The variable `environment` is bound to the environment in which the closure was created. And not to the current environment.



Dynamic scoping with SmallScheme

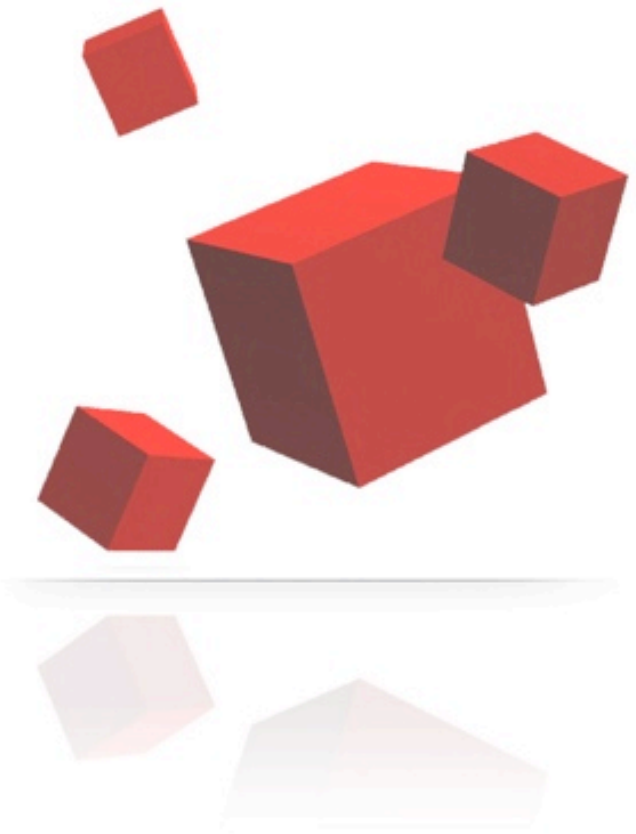
```
SchemePrimitiveFunction>>
   apply: listOfArguments withEnv: anEnvironment
   | env b valuesProvided argNames |
   valuesProvided := listOfArguments.
   argNames := arguments.
   env := SchemeEnvironment createFromEnvironment: anEnvironment.
   ...
```

The variable `anEnvironment` is bound to the current environment. Which is not the environment in which the closure was created.



Further notes

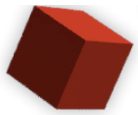
1. Symbiosis
2. Tail recursion
3. Grammar
4. Useful links



Symbiosis between Smalltalk and Scheme

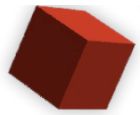
Few class extensions:

```
Object>>asSmalltalkObject
  ^ self

SchemeObject>>asSmalltalkObject
  self subclassResponsibility

SchemeSymbol>>asSmalltalkObject
  ^ self value

...
```

Tail Recursion (R5RS, Section 3.5)

- But... We did not talk about tail recursion...
- "Tail-recursion allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."
- Easy to implement in SmallScheme



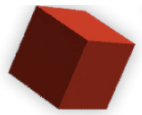
Tail Recursion

Intuitively there is a stack consumption in:

```
(define (fac n)
  (if (= n 0)
    1
    (* n (fac (- n 1)))))
```

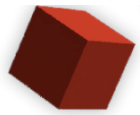
But there is no stack consumption in:

```
(define (fac n) (fac2 n 1))

(define (fac2 n acc)
  (if (= n 0)
     acc
     (fac2 (- n 1) (* acc n))))
```

Minimal or large grammar?

- SmallScheme uses a very minimal grammar.
- The fact that we use a small grammar:
  - new primitives can be added without having to modify the grammar
  - primitive names can be easily changed
- However we need:
  - evaluation of arguments need to be controlled (evaluateArguments var in primitive)
  - need to have a MakeClosure primitive



Useful links

SmallScheme:

`http://www.squeaksource.com/Scheme`

Smacc (also available on SqueakMap):

`http://www.refactory.com/Software/SmaCC`

Squeak:

`http://www.squeak.org`

Everything you wish to know about Scheme:

`http://www.schemers.org`

Especially the R5RS:

`http://www.schemers.org/Documents/Standards/R5RS/`