Lenguajes de **P**rogramación

- First-Order Functions

Federico Olmedo
Ismael Figueroa

Adding functions to our language

We want to extend our language with the ability to define and apply functions.

Indeed, most programming languages allow developers to abstract behavior by means of functions (or methods, subroutines, etc) in order to improve modularity and program abstraction.

Although we would like to have functions as values, as in Racket, we will start with a simpler approach: the so-called ***first-order functions***.

We want to run programs such as:

```
{+ 1 {double 2}}
```

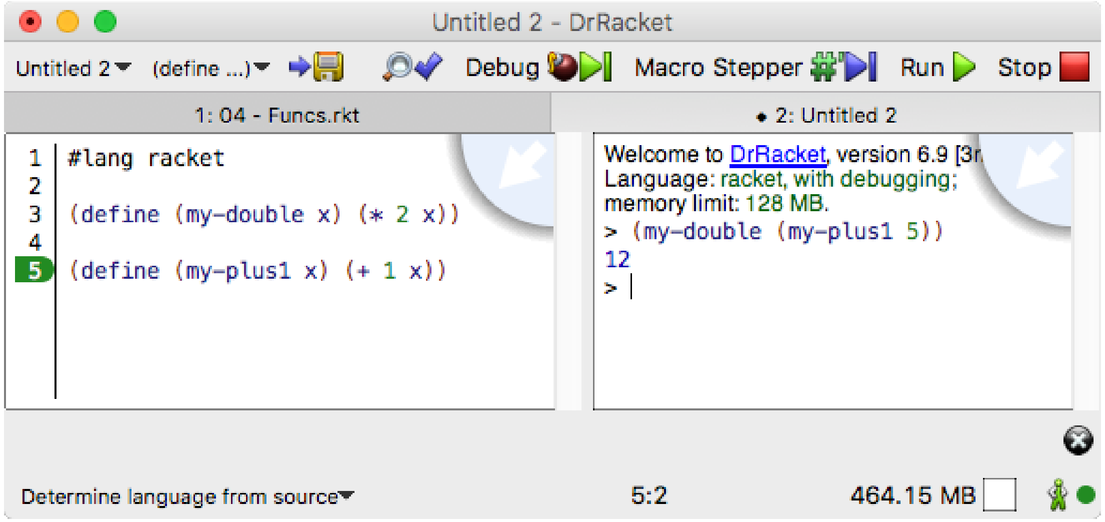
Where `double` is a function defined elsewhere.

3 Modeling approach

Conceptually, the approach is similar to DrRacket:

function(s) definition + expression to evaluate

(window on the left) (window on the right)



The idea is that as function definitions are available to be applied in the REPL, in our language function definitions are available to be applied in users' programs.

Modeling approach

First-order functions

- Functions are <u>not</u> values
- <u>Cannot be anonymous</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

We want to run programs such as:

```
{+ 1 {double 2}}
```

Where `double` is a function defined elsewhere as:

```
{fundef 'double 'n (parse '{+ n n})}
```

Modeling approach: defining functions

We want to run programs such as: `{+ 1 {double 2}}`

Where **`double`** is a function defined elsewhere as:

```
{fundef 'double 'n (parse '{+ n n})}
```

Modeling approach: defining functions

We want to run programs such as:

```
{+ 1 {double 2}}
```

Where `double` is a function defined elsewhere as:

```
{fundef 'double 'n (parse '{+ n n})}
```

function definition structure constructor

Modeling approach: defining functions

We want to run programs such as: `{+ 1 {double 2}}`

Where **`double`** is a function defined elsewhere as:

```
{fundef 'double 'n (parse '{+ n n})}
```

function name

function definition structure constructor

Modeling approach: defining functions

We want to run programs such as: `{+ 1 {double 2}}`

Where **`double`** is a function defined elsewhere as:

function name

```
{fundef 'double 'n (parse '{+ n n})}
```

function definition structure constructor

parameter name

Modeling approach: defining functions

We want to run programs such as: `{+ 1 {double 2}}`

Where `double` is a function defined elsewhere as:

function name

```
{fundef 'double 'n (parse '{+ n n})}
```

function definition structure constructor

parameter name

function body

Modeling approach: defining functions

We want to run programs such as: `{+ 1 {double 2}}`

Where **`double`** is a function defined elsewhere as:

function name

```
{fundef 'double 'n (parse '{+ n n})}
```

function definition structure constructor

parameter name

function body

A function is defined by its name, the name of its parameter, and its body. <u>The body is a proper AST</u> (using Expr deftype).

Modeling approach: defining functions

We want to run programs such as: `{+ 1 {double 2}}`

Where **`double`** is a function defined elsewhere as:

function name

```
{fundef 'double 'n (parse '{+ n n})}
```

function definition structure constructor

parameter name

function body

A function is defined by its name, the name of its parameter, and its body. The body is a proper AST (using Expr deftype).

As a sub-program. We must determine what is the semantics of interpreting function application. We will see this depends on substitution.

How do we extend our language semantics?

EXTEND CONCRETE SYNTAX:

how will developers define functions?
how will developers apply functions?

EXTEND ABSTRACT SYNTAX:

what information will we extract for function definition? and for function application?

EXTEND PARSER:

what elements of the parser we must update?
also update the definition of the substitution function

EXTEND INTERPRETER:

what is the semantics of applying first-order functions?

Extending concrete and abstract syntax

Similar to Racket, we assume any expression that starts with an identifier is a function application.

Extend Concrete Syntax

```
#|
<s-expr> ::= <num>
           | (list '+ <s-expr> <s-expr>)
           | (list '- <s-expr> <s-expr>)
           | (list 'if0 <s-expr> <s-expr> <s-expr>)
           | (list 'with (list <sym> <s-expr>) <s-expr>)
           | <sym>
           | (list <sym> <s-expr>)
|#
```

Extend Abstract Syntax

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
         | (if0 <expr> <expr> <expr>)
         | (with <sym> <expr> <expr>)
         | (id <sym>)
         | (app <sym> <expr>)
|#
```

We need additional syntax for function definitions:

```
#|
<fundef> ::= (fundef <sym> <sym> <expr>)
|#
```

Note function bodies are full-blown ASTs. They cannot contain other function definitions.

Following the grammars: abstract syntax

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
         | (if0 <expr> <expr> <expr>)
         | (with <sym> <expr> <expr>)
         | (id <sym>)
         | (app <sym> <expr>)
|#
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (with x e b)
  (id x)
  (app f-name f-arg))
```

Following the grammars: concrete syntax & parser

```
#|
<s-expr> ::= <num>
           | (list '+ <s-expr> <s-expr>)
           | (list '- <s-expr> <s-expr>)
           | (list 'if0 <s-expr> <s-expr> <s-expr>)
           | (list 'with (list <sym> <s-expr>) <s-expr>)
           | <sym>
           | (list <sym> <s-expr>)
|#
(define (parse s-expr)
  (match s-expr
    [(? number? n) (num n)]
    [(? symbol? x) (id x)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'if0 c t f) (if0 (parse c) (parse t) (parse f))]
    [(list 'with (list x e) b) (with x (parse e) (parse b))]
    [(list f-name f-arg) (app f-name (parse f-arg))]))
```

Following the grammars: function definitions

```
#|
<fundef> ::= (fundef <sym> <sym> <expr>)
|#
(deftype FunDef
  (fundef name arg body))
```

```
;; lookup :: symbol listof(FunDef) -> FunDef
;; Searches a function definition within a list of definitions.
(define (lookup f-name defs)
  (match defs
    [(list) (error 'lookup "Function ~a not found" f-name)]
    [(cons head tail) (if (symbol=? f-name (fundef-name head))
                          head
                          (lookup f-name tail))]))
```

Updating the substitution function

Applying substitution to function application nodes simply means to propagate the substitution into the argument expression.

```
;; subst :: Expr Symbol Expr -> Expr
;; (subst in what for) substitutes all free occurrences
;; of identifier 'what' in expression 'in' for expression 'for'.
(define (subst in what for)
  (match in
    [(num n) (num n)]
    [(add l r) (add (subst l what for) (subst r what for))]
    [(sub l r) (sub (subst l what for) (subst r what for))]
    [(if0 c t f) (if0 (subst c what for) (subst t what for) (subst f what for))]
    [(with x e b) (with x (subst e what for) (if (eq? x what) b (subst b what for)))]
    [(id x) (if (eq? x what) for (id x))]
    [(app f-name f-arg) (app f-name (subst f-arg what for))]))
```

Putting it all together

We want to run programs such as:

```
{+ 1 {double 2}}
```

Where **double** is a function defined elsewhere as:

```
{fundef 'double 'n (parse '{+ n n})}
```

```
;; interp :: Expr listof(FunDef) -> number
;; Evaluates arithmetic expression with function calls
(define (interp expr f-list) ...)
```

```
(interp (parse '{+ 1 {double 2}})
        (list (fundef 'double 'n (parse '{+ n n}))))
```

Program results depend on the available function definitions

The interpreter

```
;; interp :: Expr listof(FunDef) -> number
;; Evaluates arithmetic expression with function calls
(define (interp expr f-list)
  (match expr
    [(num n) n]
    [(add l r) (+ (interp l f-list) (interp r f-list))]
    [(sub l r) (- (interp l f-list) (interp r f-list))]
    [(if0 c t f) (if (zero? (interp c f-list)) (interp t f-list) (interp f f-list))]
    [(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
    [(id x) (error 'interp "Open expression (free occurrence of ~a)" x)]
    [(app f-name f-arg)

                                                                          ]))
```

The interpreter

```
;; interp :: Expr listof(FunDef) -> number
;; Evaluates arithmetic expression with function calls
(define (interp expr f-list)
  (match expr
    [(num n) n]
    [(add l r) (+ (interp l f-list) (interp r f-list))]
    [(sub l r) (- (interp l f-list) (interp r f-list))]
    [(if0 c t f) (if (zero? (interp c f-list)) (interp t f-list) (interp f f-list))]
    [(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
    [(id x) (error 'interp "Open expression (free occurrence of ~a)" x)]
    [(app f-name f-arg)
  1  (def (fundef _ the-arg the-body) (lookup f-name f-list))
                                                         ]))
```

1 First, we extract the name of the function argument, as well as the function body.

The interpreter

```
;; interp :: Expr listof(FunDef) -> number
;; Evaluates arithmetic expression with function calls
(define (interp expr f-list)
  (match expr
    [(num n) n]
    [(add l r) (+ (interp l f-list) (interp r f-list))]
    [(sub l r) (- (interp l f-list) (interp r f-list))]
    [(if0 c t f) (if (zero? (interp c f-list)) (interp t f-list) (interp f f-list))]
    [(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
    [(id x) (error 'interp "Open expression (free occurrence of ~a)" x)]
    [(app f-name f-arg)
   ① (def (fundef _ the-arg the-body) (lookup f-name f-list))
                                   ② (num (interp f-arg f-list))         ]))
```

1. First, we extract the name of the function argument, as well as the function body.
2. Interpret the function argument to get its value, which is reinstated as AST node.

The interpreter

```
;; interp :: Expr listof(FunDef) -> number
;; Evaluates arithmetic expression with function calls
(define (interp expr f-list)
  (match expr
    [(num n) n]
    [(add l r) (+ (interp l f-list) (interp r f-list))]
    [(sub l r) (- (interp l f-list) (interp r f-list))]
    [(if0 c t f) (if (zero? (interp c f-list)) (interp t f-list) (interp f f-list))]
    [(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
    [(id x) (error 'interp "Open expression (free occurrence of ~a)" x)]
    [(app f-name f-arg)
  ❶ (def (fundef _ the-arg the-body) (lookup f-name f-list))
         ❸(subst the-body the-arg (num (interp f-arg f-list)))       ]))
                                          ❷
```

1. First, we extract the name of the function argument, as well as the function body.
2. Interpret the function argument to get its value, which is reinstated as AST node.
3. Substitute free occurrences of the argument name, in the function body, for the computed value.

The interpreter

```
;; interp :: Expr listof(FunDef) -> number
;; Evaluates arithmetic expression with function calls
(define (interp expr f-list)
  (match expr
    [(num n) n]
    [(add l r) (+ (interp l f-list) (interp r f-list))]
    [(sub l r) (- (interp l f-list) (interp r f-list))]
    [(if0 c t f) (if (zero? (interp c f-list)) (interp t f-list) (interp f f-list))]
    [(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
    [(id x) (error 'interp "Open expression (free occurrence of ~a)" x)]
    [(app f-name f-arg)
   1 (def (fundef _ the-arg the-body) (lookup f-name f-list))
   4 (interp (subst the-body the-arg (num (interp f-arg f-list))) f-list)]))
                3                                  2
```

1. First, we extract the name of the function argument, as well as the function body.
2. Interpret the function argument to get its value, which is reinstated as AST node.
3. Substitute free occurrences of the argument name, in the function body, for the computed value.
4. Interpret the function body after the argument substitution.

Example

```
;; run :: s-expr listof(FunDef) -> number
(define (run prog f-list)
  (interp (parse prog) f-list))

(define my-funcs
  (list (fundef 'double 'x (parse '{+ x x}))
        (fundef 'plus1 'n (parse '{+ n 1}))))

(test (run '{+ 1 {double (plus1 4)}} my-funcs) 11)
```

Example

```
;; run :: s-expr listof(FunDef) -> number
(define (run prog f-list)
  (interp (parse prog) f-list))

(define my-funcs
  (list (fundef 'double 'x (parse '{+ x x}))
        (fundef 'plus1 'n (parse '{+ n 1}))))

(test (run '{+ 1 {double (plus1 4)}} my-funcs) 11)
```

Using our arithmetical language define function my-S

my-S(0) = 0   my-S(n+1) = n+1 + my-S(n)

that sums up the first n natural numbers.

Some observations about our interpreter

Some observations about our interpreter

- A function might be defined multiple times. By our definition of `lookup`, the first (left-most in the list) definition takes precedence.

Some observations about our interpreter

- A function might be defined multiple times. By our definition of `lookup`, the first (left-most in the list) definition takes precedence.

- Functions are accessible to other functions, and moreover,

Some observations about our interpreter

- A function might be defined multiple times. By our definition of `lookup`, the first (left-most in the list) definition takes precedence.

- Functions are accessible to other functions, and moreover,

- Functions may call one another, regardless of the order in which they are defined.

Lecture material

Bibliography

- Programming Languages: Application and Interpretation (1st Edition)
  Shriram Krishnamurthi [Download]
  Chapters 3, 4

Source code

- Arithmetical language with function calls [Download]

Lenguajes de Programación

- Evaluation by deferred substitution
- Static and dynamic scoping

Federico Olmedo
Ismael Figueroa

- Evaluation by deferred substitution

Recap: language of arithmetical expressions + functions

```
(deftype Expr
  (num n)                          — constants
  (add l r)
  (sub l r)                        — addition, subtraction
  (if0 c t f)                      — conditionals
  (with id named-expr body)
  (id x)                           — local definitions
  (app f-name f-arg))              — function application
```

Recap: language of arithmetical expressions + functions

Our interpreter:

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

— trivial

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

— trivial

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

— trivial

— simply combining the value of subexpressions

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

— trivial

— simply combining the value of subexpressions

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

— trivial

— simply combining the value of subexpressions

— uses substitution

Recap: language of arithmetical expressions + functions

Our interpreter:

How does it work?

```
(define (interp expr f-list)
  (match expr

    [(num n) ...]

    [(add l r) ...]
    [(sub l r) ...]
    [(if0 c t f) ...]

    [(with id named-expr body) ...]
    [(app f-name f-arg) ...]

    [(id x) ...]

    ))
```

— trivial

— simply combining the value of subexpressions

— uses substitution

— error (open expression)

Evaluation by substitution

Local definitions and function calls are handled using substitution

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.
3. Reduce the obtained expression.

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.
3. Reduce the obtained expression.

```
[(app f-name f-arg)
 (def (fundef _ the-arg the-body) (lookup f-name f-list))
 (interp (subst the-body the-arg (num (interp f-arg f-list))) f-list)]))
```

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.
3. Reduce the obtained expression.

```
[(app f-name f-arg)
 (def (fundef _ the-arg the-body) (lookup f-name f-list))
 (interp (subst the-body the-arg (num (interp f-arg f-list))) f-list)]))
```

1. Retrieve argument name **the-arg** and body **the-body** from the function definition.

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.
3. Reduce the obtained expression.

```
[(app f-name f-arg)
 (def (fundef _ the-arg the-body) (lookup f-name f-list))
 (interp (subst the-body the-arg (num (interp f-arg f-list))) f-list)]))
```

1. Retrieve argument name `the-arg` and body `the-body` from the function definition.
2. Reduce actual argument `f-arg` recursively.

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.
3. Reduce the obtained expression.

```
[(app f-name f-arg)
 (def (fundef _ the-arg the-body) (lookup f-name f-list))
 (interp (subst the-body the-arg (num (interp f-arg f-list))) f-list)]))
```

1. Retrieve argument name **the-arg** and body **the-body** from the function definition.
2. Reduce actual argument **f-arg** recursively.
3. Substitute the formal argument in the function body, with the computed value.

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.
3. Reduce the obtained expression.

```
[(app f-name f-arg)
 (def (fundef _ the-arg the-body) (lookup f-name f-list))
 (interp (subst the-body the-arg (num (interp f-arg f-list))) f-list)]))
```

1. Retrieve argument name **the-arg** and body **the-body** from the function definition.
2. Reduce actual argument **f-arg** recursively.
3. Substitute the formal argument in the function body, with the computed value.
4. Reduce the obtained expression.

Evaluation by substitution

Local definitions and function calls are handled using substitution

```
[(with x e b) (interp (subst b x (num (interp e f-list))) f-list)]
```

1. Reduce the named expression **e** recursively.
2. Substitute in body **b** identifier **x** with the value of the named expression **e**.
3. Reduce the obtained expression.

ADDS EVALUATION OVERHEAD

```
[(app f-name f-arg)
 (def (fundef   the-arg the-body) (lookup f-name f-list))
 (interp (subst the-body the-arg (num (interp f-arg f-list)))        ist)]))
```

1. Retrieve argument name **the-arg** and body **the-body** from the function definition.
2. Reduce actual argument **f-arg** recursively.
3. Substitute the formal argument in the function body, with the computed value.
4. Reduce the obtained expression.

Evaluation by substitution

Local definitions and function calls are handled using substitution. This substitution adds evaluation overhead.

Evaluation by substitution

Local definitions and function calls are handled using substitution. This substitution adds evaluation overhead.

For example, to reduce expression

```
{with {x 3} {with {y 4} {with {z 5} {+ x {+ y z}}}}}
```

Evaluation by substitution

Local definitions and function calls are handled using substitution. This substitution adds evaluation overhead.

For example, to reduce expression

```
{with {x 3} {with {y 4} {with {z 5} {+ x {+ y z}}}}}
```

the interpreter needs to performs the following substitutions:

```
(subst  {with {y 4} {with {z 5} {+ x {+ y z}}}} x 3)
```

Evaluation by substitution

Local definitions and function calls are handled using substitution. This substitution adds evaluation overhead.

For example, to reduce expression

```
{with {x 3} {with {y 4} {with {z 5} {+ x {+ y z}}}}}
```

the interpreter needs to performs the following substitutions:

```
(subst  {with {y 4} {with {z 5} {+ x {+ y z}}}} x 3)
(subst  {with {z 5} {+ x {+ y z}}} y 4)
```

Evaluation by substitution

Local definitions and function calls are handled using substitution. This substitution adds evaluation overhead.

For example, to reduce expression

```
{with {x 3} {with {y 4} {with {z 5} {+ x {+ y z}}}}}
```

the interpreter needs to performs the following substitutions:

```
(subst  {with {y 4} {with {z 5} {+ x {+ y z}}}} x 3)
(subst  {with {z 5} {+ x {+ y z}}} y 4)
(subst  {+ 3 {+ 4 z}} z 5)
```

Evaluation by substitution

Local definitions and function calls are handled using substitution. This substitution adds evaluation overhead.

For example, to reduce expression

```
{with {x 3} {with {y 4} {with {z 5} {+ x {+ y z}}}}}
```

the interpreter needs to performs the following substitutions:

```
(subst  {with {y 4} {with {z 5} {+ x {+ y z}}}} x 3)
(subst  {with {z 5} {+ x {+ y z}}} y 4)
(subst  {+ 3 {+ 4 z}} z 5)
```

which in turn requires “walking through” the three grayed expressions

Evaluation by deferred substitution

<u>We can do better</u> by deferring the substitutions instead of performing them right after encountering a local definition (or function application).

Evaluation by deferred substitution

We can do better by deferring the substitutions instead of performing them right after encountering a local definition (or function application).

To do so we make use of a repository of deferred substitutions during evaluation:

Evaluation by deferred substitution

We can do better by deferring the substitutions instead of performing them right after encountering a local definition (or function application).

To do so we make use of a repository of deferred substitutions during evaluation:

- Start up with an empty repository

Evaluation by deferred substitution

We can do better by deferring the substitutions instead of performing them right after encountering a local definition (or function application).

To do so we make use of a repository of deferred substitutions during evaluation:

- Start up with an empty repository
- When finding a substitution (originated by a local definition or function application), instead of doing it, we add an entry to the repository associating the identifier with its value and continue the evaluation.

Evaluation by deferred substitution

We can do better by deferring the substitutions instead of performing them right after encountering a local definition (or function application).

To do so we make use of a repository of deferred substitutions during evaluation:

- Start up with an empty repository
- When finding a substitution (originated by a local definition or function application), instead of doing it, we add an entry to the repository associating the identifier with its value and continue the evaluation.
- Upon the occurrence of an identifier, we retrieve its value from the repository.

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))

(define extend-env aEnv)
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))

(define extend-env aEnv)

(define (env-lookup x env)
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))

(define extend-env aEnv)

(define (env-lookup x env)
  (match env
    [(mtEnv)                                                   ]
    [(aEnv id val rest)

                                                   ]))
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))

(define extend-env aEnv)

(define (env-lookup x env)
  (match env
    [(mtEnv)                                          ]
    [(aEnv id val rest) (if (symbol=? id x)

                                                 ]))
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))

(define extend-env aEnv)

(define (env-lookup x env)
  (match env
    [(mtEnv)                                                    ]
    [(aEnv id val rest) (if (symbol=? id x)
                            val
                                                  ]))
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))

(define extend-env aEnv)

(define (env-lookup x env)
  (match env
    [(mtEnv)                                                  ]
    [(aEnv id val rest) (if (symbol=? id x)
                            val
                            (env-lookup x rest))]))
```

Modeling the repository of deferred substitutions

— interface of the ADT

```
;; empty-env  :: Env
;; extend-env :: Symbol Value Env -> Env
;; env-lookup :: Symbol Env -> Value
```

— implementation of the ADT

```
;; <env> ::= mtEnv
;;         | (aEnv <id> <value> <env>)
(deftype Env
  (mtEnv)
  (aEnv id val env))

(define empty-env (mtEnv))

(define extend-env aEnv)

(define (env-lookup x env)
  (match env
    [(mtEnv) (error 'env-lookup "free identifier: ~a" x)]
    [(aEnv id val rest) (if (symbol=? id x)
                            val
                            (env-lookup x rest))]))
```

Adapting the interpreter

```
;; interp :: Expr listof(FunDef) Env -> number
;; Evaluates an arithmetic expression with function calls
;; and local definitions deferring the substitutions.
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

Adapting the interpreter

```
;; interp :: Expr listof(FunDef) Env -> number
;; Evaluates an arithmetic expression with function calls
;; and local definitions deferring the substitutions.
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

— Evaluate the body of the local definition in the augmented environment (mapping the identifier with the value of the named expression)

Adapting the interpreter

```
;; interp :: Expr listof(FunDef) Env -> number
;; Evaluates an arithmetic expression with function calls
;; and local definitions deferring the substitutions.
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

— Evaluate the body of the local definition in the augmented environment (mapping the identifier with the value of the named expression)

Adapting the interpreter

```
;; interp :: Expr listof(FunDef) Env -> number
;; Evaluates an arithmetic expression with function calls
;; and local definitions deferring the substitutions.
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

— Evaluate the body of the local definition in the augmented environment (mapping the identifier with the value of the named expression)

```
    [(id x) (env-lookup x env)]
```

— Retrieve the value of the identifier from the environment

Adapting the interpreter

```
;; interp :: Expr listof(FunDef) Env -> number
;; Evaluates an arithmetic expression with function calls
;; and local definitions deferring the substitutions.
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

— Evaluate the body of the local definition in the augmented environment (mapping the identifier with the value of the named expression)

```
    [(id x) (env-lookup x env)]
```

— Retrieve the value of the identifier from the environment

```
    [(app f e) (def (fundef _ the-arg the-body) (lookup f f-list))
               (def new-env (extend-env the-arg (interp e f-list env) empty-env))
               (interp the-body f-list new-env)]))
```

— Evaluate the function's body in the environment only mapping the formal parameter with the value of the actual parameter

Adapting the interpreter

```
;; interp :: Expr listof(FunDef) Env -> number
;; Evaluates an arithmetic expression with function calls
;; and local definitions deferring the substitutions.
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

— Evaluate the body of the local definition in the augmented environment (mapping the identifier with the value of the named expression)

```
    [(id x) (env-lookup x env)]
```

— Retrieve the value of the identifier from the environment

More to come later

```
    [(app f e) (def (fundef _ the-arg the-body) (lookup f f-list))
               (def new-env (extend-env the-arg (interp e f-list env) empty-env))
               (interp the-body f-list new-env)]))
```

— Evaluate the function's body in the environment only mapping the formal parameter with the value of the actual parameter

Some remarks on the new interpreter

Some remarks on the new interpreter

- The interpreter with deferred substitution, and the original interpreter with explicit substitution return both the same output

- For this to hold true, it is indispensable that when querying the repository for an identifier with multiple bindings, it returns <u>**the most recently added**</u>.

Some remarks on the new interpreter

- The interpreter with deferred substitution, and the original interpreter with explicit substitution return both the same output

- For this to hold true, it is indispensable that when querying the repository for an identifier with multiple bindings, it returns **the most recently added**.

Under this policy: `{with {x 0} {with {x 1} x}}` $\rightsquigarrow$ `1`

Some remarks on the new interpreter

```
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

Why do we never remove entries from the repository of deferred substitutions?

Some remarks on the new interpreter

```
(define (interp expr f-list env)
  (match expr

    [(with x e b) (def new-env (extend-env x (interp e f-list env) env))
                  (interp b f-list new-env)]
```

What happens if when evaluating a local definition, we use a fresh repository, bounding only the identifier at hand (as in the case of function application)?

Why do we never remove entries from the repository of deferred substitutions?

- Static and dynamic scoping

Static vs dynamic scope

Consider expression

```
{with {n 5} {f 10}}
```

with function definition

```
(list (fundef 'f 'x (id 'n)))
```

To which value should it reduce?

Static vs dynamic scope

Consider expression

```
{with {n 5} {f 10}}
```

with function definition

```
(list (fundef 'f 'x (id 'n)))
```

To which value should it reduce?

- **5**
- Error (n is free)

Static vs dynamic scope

Consider expression

```
{with {n 5} {f 10}}
```

with function definition

```
(list (fundef 'f 'x (id 'n)))
```

To which value should it reduce?

- 5
- Error (n is free)

Our interpreter(s)

Static vs dynamic scope

Consider expression

```
{with {n 5} {f 10}}
```

with function definition

```
(list (fundef 'f 'x (id 'n)))
```

To which value should it reduce?

- **5** — **DYNAMIC SCOPE**
- Error (n is free) — **STATIC (OR LEXICAL) SCOPE**

Our interpreter(s)

Scoping rules determine how identifiers obtain their values

Static scope

In a language with *static scope*, the scope of a binding is a syntactically delimited region.

```
{with {n 5} {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'n)))
```

scope of the binding `n => 5`

Static scope

In a language with *static scope*, the scope of a binding is a syntactically delimited region.

`{with {n 5} {f 10}}` w/ def. `(list (fundef 'f 'x (id 'n)))`

scope of the binding `n => 5`

The binding `n => 5` has effect only in “text” `{f 10}`.
As there is no occurrence of `n` in `{f 10}`, the binding has indeed no effect.

Static scope

In a language with *static scope*, the scope of a binding is a syntactically delimited region.

`{with {n 5} {f 10}}` w/ def. `(list (fundef 'f 'x (id 'n)))`

scope of the binding `n => 5`

The binding `n => 5` has effect only in “text” `{f 10}`.
As there is no occurrence of `n` in `{f 10}`, the binding has indeed no effect.

- Corresponds to the semantics provided by either of our interpreters

Static scope

In a language with *static scope*, the scope of a binding is a syntactically delimited region.

`{with {n 5} {f 10}}` w/ def. `(list (fundef 'f 'x (id 'n)))`

scope of the binding `n => 5`

The binding `n => 5` has effect only in "text" `{f 10}`.
As there is no occurrence of `n` in `{f 10}`, the binding has indeed no effect.

- Corresponds to the semantics provided by either of our interpreters
- Used in most modern programming languages: Ada, C, Pascal, Scheme, and Haskell, Python, etc.

Dynamic scope

In a language with *dynamic scope*, the scope of a binding is the entire remaining of the execution during which the binding is in effect.

`{with {n 5} {f 10}}` w/ def. `(list (fundef ‘f ‘x (id ’n)))`

Dynamic scope

In a language with *dynamic scope*, the scope of a binding is the entire remaining of the execution during which the binding is in effect.

`{with {n 5} {f 10}}` w/ def. `(list (fundef ‘f ‘x (id ’n)))`

The binding `n => 5` still has effect during the “execution” of **`f`**.
And during the “execution” of any other function that f invokes.

Dynamic scope

Providing a dynamic scope to our language requires adapting only the evaluation rule of function calls to:

```
(define (interp expr f-list env)
  (match expr
    [(app f e) (def (fundef _ the-arg the-body) (lookup f f-list))
               (def new-env (extend-env the-arg (interp e f-list env) env))
               (interp the-body f-list new-env)]))
```

i.e. for evaluating the function's body, the current environment must be augmented (instead of discarded) with the binding of the function's formal argument

Dynamic scope

∴

Dynamic scope

- The context where a function is invoked passes over bindings

Dynamic scope

- The context where a function is invoked passes over bindings
- The same function invocation within two different contexts may reduce to different values.

Dynamic scope

- The context where a function is invoked passes over bindings
- The same function invocation within two different contexts may reduce to different values.

> **[…] dynamic scope is entirely unreasonable. The problem is that we simply cannot determine what the value of a program will be without knowing everything about its execution history […] We will therefore regard dynamic scope as an error and reject its use.**
>
> **PLAI, Shriram Krishnamurthi**

Static vs dynamic scope

<u>Scoping rules determine how identifiers obtain their values</u>

Static scope

- Based on the **<u>structure of the code</u>** as it is written.
- The scope is determined when the code is written.
- It is predictable. We just have to read the source code.
- The proper default for modern languages.
- Generally has better tooling support and scalability.

Dynamic scope

- Based on the **<u>history of function calls</u>** during program execution.
- The scope is determined at runtime, based on the call stack.
- It's difficult to predict, resulting in weird bugs.
- Useful as an optional and explicit feature.
- Can be used to pass arbitrary parameters without changing function signatures.

Dynamic scope

Is it such an error? No!

Dynamic scope

Is it such an error? No!

- In certain cases, it provides some desired flexibility
  - Configuration: e.g. system-wise STDOUT
  - Adaptation: ability to behave differently according to the context (e.g. desktop vs mobile)

See A Note on Dynamic Scope by Eric Tanter !!!

Dynamic scope

Is it such an error? No!

- In certain cases, it provides some desired flexibility
  - Configuration: e.g. system-wise STDOUT
  - Adaptation: ability to behave differently according to the context (e.g. desktop vs mobile)

See A Note on Dynamic Scope by Eric Tanter !!!

- It is good as an opt-in feature (as in Common Lisp, Scala, Racket.), **but not as the default option.**

Lecture material

Bibliography

- Programming Languages: Application and Interpretation (1st Edition)
  Shriram Krishnamurthi [Download]
  Chapter 5
- A Note on Dynamic Scope by Éric Tanter

Source code

- Arithmetical language with deferred substitution [Download]